Caractérisation et détection de malware Android basées sur les flux d'information.

Radoniaina Andriatsimandefitra Ratsisahanana

N° d'ordre : 2014-25-TH

SUPELEC

Ecole Doctorale MATISSE
« Mathématiques, Télécommunications, Informatique, Signal, Systèmes Electroniques »

THÈSE DE DOCTORAT

DOMAINE : STIC
Spécialité : Informatique

Préparée au sein de l'équipe projet CIDRE
Confidentialité, Intégrité, Disponibilité, Répartition
Supélec / INRIA

Soutenue le 15 décembre 2014

par :

Radoniaina ANDRIATSIMANDEFITRA RATSISAHANANA

Caractérisation et détection de malware Android basées sur les flux d'information

Directeur de thèse :	Ludovic MÉ	Professeur à Supélec
Encadrant de thèse :	Valérie VIET TRIEM TONG	Professeur associé à Supélec
Composition du jury :		
Président du jury :	David PICHARDIE	Professeur à l'ENS Rennes
Rapporteurs :	Jean-Yves MARION	Professeur à l'université de Lorraine
	Radu STATE	Chercheur à l'université du Luxembourg
Examinateurs :	Pascal CARON	Maître de Conférence à l'université de Rouen
	Ludovic MÉ	Professeur à Supélec
	Valérie Viet Triem Tong	Professeur associé à Supélec
Membres invités :	Anthony Desnos	Ingénieur chez Google France

Table des matières

Table des figures

Liste des listings

Liste des tableaux

Liste des algorithmes

Remerciements

Avant d'aborder le pourquoi de ce document, je tenais avant tout à remercier les membres du jury d'avoir accepté d'évaluer mon travail : David Pichardie pour avoir accepté de présider le jury de ma thèse, Radu State et Jean-Yves Marion pour avoir accepté d'être les rapporteurs de ce mémoire, Pascal Caron et Anthony Desnos pour avoir accepté d'examiner mon travail. Je tiens à remercier spécialement Ludovic Mé et Valérie Viet Triem Tong pour leur encadrement et leur implication dans la réalisation de ma thèse, même si parfois cela n'a pas été facile, ainsi que le reste de l'équipe pour ces trois années passées ensemble. Finalement, un grand merci aux membres de ma famille et aux amis qui m'ont soutenu tout au long de la thèse. *Kudos* au Langomatic (organisateurs, participants et staff du O'Connell's) pour toutes ces soirées du lundi riches en rencontres et en diversités culturelles.

Chapitre 1

Introduction

Lancé officiellement en 2008, Android est devenu en quelques années le système d'exploitation le plus répandu sur les plateformes mobiles de type smartphone et tablette [72, 107, 71]. Au delà de sa large adoption par le grand public, il a également suscité l'intérêt des développeurs d'applications malveillantes qui voient dans le système Android, une cible potentielle d'attaque au même niveau que les ordinateurs de bureau à cause de la diversité des données et services qu'ils proposent. Les appareils tournant sous Android offrent différentes fonctionnalités allant du simple téléphone à celles des ordinateurs de bureau et assistants numériques personnels (pockets PC ou PDA). La combinaisons de toutes ces fonctionnalités font de ces appareils un point de concentration de divers données et services sensibles (liste de contact, messages, données de géolocalisation, etc.) et en conséquence une cible de valeur pour les développeurs de malware. Ces dernières années, nous avons ainsi vu l'apparition d'un nombre toujours grandissant d'applications malveillantes qui cherchent à voler les données du téléphone, les corrompre, espionner l'utilisateur, abuser des services offerts par le téléphone, etc. À la vue du nombre grandissant des malware Android, il devient nécessaire de développer des outils d'analyse de malware afin de comprendre leur fonctionnement et plus tard les détecter.

L'une des raisons de la prolifération de ces malware est l'insuffisance des mécanismes de sécurité Android à détecter et bloquer de telles attaques et la facilité d'accès à une partie des ressources sensibles du téléphone. Android propose un ensemble de méthodes pour accéder ces ressources et ces accès nécessite une autorisation de la part de l'utilisateur au moment de l'installation. L'application est installée uniquement si l'utilisateur valide toutes les autorisations demandées par l'application. Toute application installée sur le téléphone a ainsi accès aux ressources dont elle a demandé l'accès sans que le système ne contrôle l'usage qui est fait des ressources. Les premiers travaux liés à la sécurité d'Android sont donc focalisés sur l'analyse des limites de la sécurité sous Android et sur une manière de les combler. Enck et al. proposent par exemple dans [47] une vérification des permissions demandées par les applications afin de s'assurer qu'elles ne soient trop dangereuses. Ce type d'approche a cependant une prin-

cipale limitation qui est de ne détecter que ce que nous savons être dangereux. Il ne permet donc pas de détecter et d'apprendre de nouvelles attaques.

Dans ce thèse nous adoptons une approche différente qui est basée sur les flux d'information. Un flux d'information décrit un transfert d'information entre deux objets. Au niveau du système d'exploitation, il décrit par exemple des communications entre deux applications ou des accès à un fichier. Au lieu de nous focaliser sur les limitations du système Android pour identifier les scénarios d'attaque et les détecter par la suite, nous proposons d'apprendre comment les attaques ont lieu en analysant directement les malware et utiliser la base de connaissance acquise durant l'apprentissage pour détecter les malware. Selon les travaux de Zhou et al. la méthode d'infection principale utilisée par les développeurs de malware est d'ajouter leur code malicieux à des applications existantes et proposer les versions infectées de ces applications en téléchargement sur les plateformes de téléchargement tels que Google Play. En supposant qu'un malware soit distribué sous la forme de différentes applications, nous pouvons supposer que les applications infectées par le même code malveillant ont partiellement un comportement similaire. En analysant les échantillons des malware et en isolant ces comportements similaires, nous pouvons ainsi à la fois apprendre le comportement malveillant d'un malware et calculer un profil pour ce malware afin de le détecter. Le reste de ce document est divisé comme suit.

Le chapitre 2 présente le contexte de ce travail et est divisé en deux parties. La première partie concerne le système Android : l'architecture du système, ses points communs et différences avec les systèmes Linux, la notion d'applications Android, l'analyse des limites du modèle de sécurité d'Android, les menaces que représentent les malware Android et les travaux essayant de combler ces limites. La deuxième partie du chapitre introduit les bases nécessaires à l'accomplissement de notre objectif. Nous y présentons dans un premier temps la notion de suivi de flux d'information et les différents niveaux existant pour observer les flux d'information. Par la suite nous présentons comment les méthodes d'apprentissage peuvent être utilisées afin de classifier et détecter des malware. Nous présentons pour cela quelques travaux connus exploitant les traces d'exécution des malware pour les classifier.

Le chapitre 3 présente Blare le moniteur de flux d'information et son portage pour le système Android. Blare est un outil de détection d'intrusion paramétré par une politique de flux d'information afin de détecter les intrusions dans un système. Ayant été développé pour les systèmes Linux, il ne prenait ainsi pas en compte une partie des flux existant sous Android. Ces flux sont liés à des mécanismes propres à Android qui n'existent pas sous Linux. Dans un premier temps, nous présentons donc ces mécanismes et les flux d'information qu'ils causent qui étaient invisibles à Blare. Ensuite, nous présentons comment nous avons pris en compte ces mécanismes et amélioré ainsi l'observation des flux d'information sous Android. Le résultat de ces améliorations, AndroBlare, est ce qui nous sert d'environnement d'analyse durant cette thèse.

Le chapitre 4 introduit une structure de donnée appelée graphe de flux système ou System Flow Graph. Cette structure décrit de manière compacte et plus compréhensible les flux d'information que Blare observe dans les systèmes. Un

des objectifs de cette thèse est de comprendre le fonctionnement d'un malware en l'analysant dans notre environnement d'analyse AndroBlare. Après avoir présenté la structure, nous montrons dans ce chapitre, à travers l'analyse d'un échantillon de malware, l'utilité de la structure pour représenter le comportement d'un malware et comprendre son comportement. Cette structure représentant le comportement d'une application, nous montrons également qu'elle peut assister un développeur dans la création d'une politique de flux d'information pour son application.

Enfin, le chapitre 5 présente notre approche afin de classifier et détecter les malware Android. La classification consiste à regrouper les échantillons d'un malware avec un comportement similaire et extaire un profil caractérisant les échantillons d'un même groupe. Cette classification s'apparente à un apprentissage non supervisé et nous présentons dans ce chapitre comment nous le réalisons puis l'évaluons avec un ensemble d'échantillons de malware. La détection consiste à utiliser les profils calculés durant la classification pour détecter d'autres échantillons de malware. Nous présentons ainsi en deuxième partie de ce chapitre comment nous utilisons AndroBlare et les profils calculés pour détecter l'exécution de malware et évaluons l'approche avec d'autres échantillons.

Chapitre 2

État de l'art

L'objectif de cette thèse est de développer une méthode afin de caractériser et détecter les malwares Android. La réalisation de cet objectif s'est faite en plusieurs étapes : le portage du moniteur de flux d'information Blare pour le système Android, la proposition d'une structure de donnée pour représenter les profils d'une application et une méthode de calcul de profil des malwares Android ainsi que de leur détection. Dans ce premier chapitre, nous introduisons dans un premier temps le système Android : son architecture, la notion d'application Android et le modèle de sécurité d'Android. Ensuite, nous montrons en section 2.3 et 2.4 les limites du mécanisme de sécurité Android ainsi qu'un aperçu des menaces que représentent les malware Android. La section 2.5 présente ensuite les différents travaux visant à combler les limites du mécanisme de sécurité d'Android ainsi qu'à détecter les attaques visant le système. L'approche pour laquelle nous avons optée est basée sur les flux d'information au niveau du système. La section 2.6 présente ainsi différents travaux liés au suivi de flux d'information et une discussion sur les apports et limitations de ces approches. Une partie des expérimentations que nous menons dans cette thèse consiste à exécuter des échantillons de malware Android et plus précisément leur code malveillant. Comme évoqué dans les travaux de Zhou et al. dans [113], certains malware vérifient un nombre de conditions avant d'exécuter leur code malveillant. Afin d'identifier ces conditions, il est nécessaire d'analyser statiquement les échantillons des malware concernés. Nous présentons ainsi en section 2.8 un ensemble d'outils d'analyse statique des applications Android.

2.1 Système Android

Android partage une base commune aux systèmes Linux qui est le noyau et un ensemble de commandes et utilitaires nécessaires. Dans cette section, nous présentons dans un premier temps l'architecture du système Android, celle de ses applications et les mécanismes de sécurité fournis avec le système.

2.1.1 Architecture du système Android

Le système Android est divisé en plusieurs couches comme illustré sur la figure 2.1. La partie la plus basse représente le cœur du système, c'est-à-dire le noyau, et le reste l'espace utilisateur.

2.1.1.1 Noyau Android

Le noyau Android est une version modifiée du noyau Linux [1] et représente le cœur du système. Le noyau est le programme servant d'interface entre les différents composants du système (périphériques, processus, fichiers etc). Parmi les modifications notables apportées dans le noyau Android, nous pouvons citer les mécanismes `binder, ashmem, wakelock, low memory killer, logger, RAM console` et `Paranoid Networking`.

`Binder` est un mécanisme de communication entre processus et d'appel de méthodes distantes. L'appel de méthodes distantes, appelé *Remote Procedure Call* en anglais, consiste à faire exécuter une méthode par une entité distante. Il est inspiré du projet `OpenBinder` [81]. Comme nous le verrons par la suite, il est un élément essentiel du fonctionnement du système Android.

`Ashmem` pour *Anonymous Shared Memory* est un mécanisme de partage de mémoire similaire à celui du noyau Linux `shm`. Il est utilisé pour partager les données entre applications Android. Parmi les "nouveautés" apportées par `ashmem` il y a par exemple l'usage de compteur pour connaître le nombre de processus faisant référence à une zone de mémoire partagée et éviter les fuites de mémoire.

`Wakelock` est un mécanisme servant à notifier le noyau de ne pas se mettre en veille. Contrairement aux systèmes Linux, le système Android essaie par défaut de se mettre en veille étant donné qu'il est destiné à tourner sur des appareils à ressources limitées. Lors d'exécution de code ne devant être interrompu, le `wakelock` est ainsi utilisé pour dire au système de rester éveillé.

`Low memory killer` est un mécanisme utilisé par le noyau pour libérer de la mémoire lorsqu'il ne reste plus assez de mémoire.

`Logger` est un mécanisme de journalisation qui écrit les évènements du système uniquement dans des zones allouées en mémoire. Contrairement aux systèmes Linux traditionnels, les évènements écrits dans le journal du système ne sont ainsi jamais écrits dans un fichier.

`RAM Console` est un mécanisme qui préserve en mémoire le contenu des évènements systèmes ajoutés par du code noyau (via la fonction `printk`) lors de la précédente exécution du système. Son contenu est accessible via le fichier `/proc/last_kmsg`. Sous Linux, le contenu du journal des évènements systèmes sont stockés de manière persistante dans les fichiers sous le répertoire `/var/log/`. Ce n'est pas le cas sous Android et en cas de crash du système par exemple, il devient impossible de récupérer les évènements qui ont amené au crash. `RAM Console` est ainsi été créé pour résoudre cette limitation du système de journalisation sous Android.

1. Noyau utilisé par les distributions de type Linux telles que Debian et Ubuntu

`Paranoid Network` est un mécanisme contrôlant l'accès des applications au réseau. Sous Linux, toute application a le droit d'utiliser les `sockets` et accéder au réseau. La `socket` est la structure de base liée aux opérations réseaux sous Linux. Sous Android, l'inverse est la règle car seule les applications avec une autorisation explicite sont autorisées à créer des sockets et communiquer sur le réseau.

2.1.1.2 Espace utilisateur

L'espace utilisateur sous Android peut être divisé en plusieurs parties comme illustré sur la figure 2.1.

La couche *Applications* renferme les applications fournies par défaut sur le téléphone ainsi que celles qui seront installées plus tard par l'utilisateur. Il s'agit des applications avec lesquelles l'utilisateur pourra interagir en général (ex : application de messagerie et gestion des contacts). Les applications Android sont principalement écrites en Java. Android donne cependant la possibilité d'écrire du code natif et de l'appeler au sein de l'application grâce à une interface appelée *Java Native Interface* dont le rôle est d'interfacer du code Java avec du code natif écrit en C.

Android framework est l'ensemble des services et ressources fournies par le système Android. Il s'agit principalement de services tournant dans quelques processus clés du système Android tels que `system_server` et `mediaserver`. La commande `service list` retourne sous Android la liste des services présents dans le système. Sous Android 2.3, la commande retourne une liste de 50 services et sous Android 4.2 68 services. Parmi ces services nous pouvons citer `Service-Manager`, *LocationManager*. `ServiceManager` recense tous les autres services tournant sur le téléphone et joue un rôle d'annuaire pour les applications souhaitant accéder à un service en particulier sur le téléphone. Lorsqu'une application de navigation souhaite par exemple connaître la localisation de l'utilisateur, elle va dans un premier temps demander au `Service-Manager` la référence de `Location Service`. Une fois cette référence récupérée, elle pourra demander à `Location Service` les données de géolocalisation. `Surfaceflinger` est un autre service avec un rôle crucial sous Android. C'est lui qui est chargé de composer et dessiner ce que les différentes applications souhaitent afficher à l'écran de l'appareil.

Android runtime est l'environnement d'exécution des applications Android. Il contient la bibliothèque Java utilisable par les applications ainsi qu'une machine virtuelle appelée Dalvik. La bibliothèque reprend une partie de la bibliothèque Java standard plus quelques bibliothèques supplémentaires propres à Android. La machine virtuelle Dalvik elle interprète le *bytecode* dans lequel les applications Android ont été compilées. Le *bytecode dalvik* est différent du *bytecode Java* d'où l'usage de Dalvik à la place des machines virtuelles Java standard. Depuis Android 4.4, une deuxième machine virtuelle, ART [85, 99], qui est livrée sous forme expérimentale avec la plateforme.

Libraries renferme les bibliothèques natives du système. Elles sont généralement utilisées par les applications natives du système. Parmi les bibliothèques

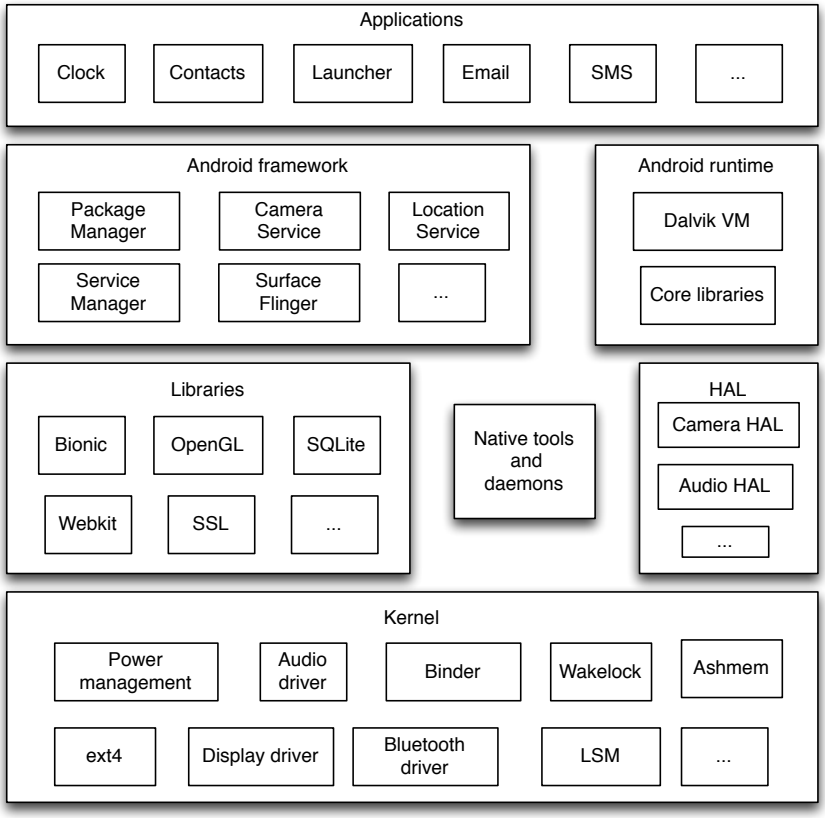


FIGURE 2.1 – Architecture de la plateforme Android

nous pouvons par exemple citer `bionic` qui renferme entre autres une bibliothèque C et une bibliothèque pour le support du C++. Selon la page Wikipédia sur `bionic` [3], il s'agit d'une version modifiée de la bibliothèque standard C de BSD et est propre à Android.

Si les applications utilisateurs sont des applications écrites en Java, il existe tout de même quelques applications natives dans le système. Il s'agit de services propres au système et de quelques outils en ligne de commande qui peuvent s'avérer utiles pour un développeur. Ces applications n'ont pas pour but d'être utilisées par l'utilisateur *lambda*.

HAL ou *Hardware Abstraction Layer* sert d'interface standard entre le système et les pilotes des périphériques présents sur l'appareil (ex : caméra, capteur etc). Les constructeurs écrivent ainsi les pilotes à leur guise mais doivent fournir les méthodes correspondant aux interfaces *HAL* pour que le système puisse utiliser les périphériques.

2.1.1.3 Communications entre processus

Android est un environnement d'exécution où les applications sont poussées à collaborer pour fonctionner. Le but de cette collaboration est de limiter la duplication de code dans les applications et de proposer un ensemble assez grand de fonctionnalités aux différentes applications du système. Un exemple illustrant cette collaboration est l'application caméra. Pour utiliser la caméra d'un appareil, une application doit connaître son mode de fonctionnement exact sachant qu'une caméra peut être différente d'un modèle de téléphone à un autre. Pour faciliter la tâche au développeur de l'application, l'application caméra du téléphone propose une interface permettant aux autres applications de prendre des photos ou enregistrer des vidéos à leur place puis leur transmettre le résultat. Ce système de collaboration repose plusieurs fonctionnalités du système. Du point de vue des développeurs d'application, la collaboration repose sur l'usage des `intents` et des composants `Content Provider` pour échanger des messages et partager du contenu. Nous donnons plus de détail sur ces mécanismes dans la section 2.1.2.1. À un niveau plus bas, la collaboration repose sur deux fonctionnalités d'Android que sont `Binder` et `Ashmem`. Comme nous l'avons écrit en section 2.1.1, ces deux mécanismes servent à effectuer des appels distants de procédure et partager des zones de mémoire entre processus.

2.1.2 Applications Android

2.1.2.1 Architecture d'une application : les différents composants

Une application Android est écrite en Java. Contrairement aux applications Java standard, une application Android peut posséder plusieurs points d'entrée. Plus précisément, une application Android peut avoir plusieurs composants et chacun d'eux peut-être un point d'entrée dans le programme. Il existe quatre types de composants : `Activity`, `ContentProvider`, `Service` et `BroadcastReceiver`.

Activity Un composant `Activity` est une interface utilisateur. Une application de messagerie électronique peut par exemple avoir trois composants `Activity` : un pour naviguer entre les répertoires de la messagerie, le deuxième pour l'affiche d'un message et le dernier pour l'édition et l'envoi.

ContentProvider Un `ContentProvider` est un composant servant au partage de données d'une application. Son rôle est de servir d'interface entre l'application souhaitant accéder aux données et les données. Ces données sont généralement stockées dans une base de données locale SQLite mais aucune restriction n'est imposée sur la manière de stocker les données. La liste des contacts est par exemple stockée dans une base de données locales dont l'accès se fait à travers un composant de type `ContentProvider`.

Service Un service est un composant effectuant des tâches en arrière plan. Il est utilisé pour effectuer de longue tâches internes à l'application ou à exécuter une tâche à la demande d'une application. Dans un client File Transfer Protocol, un service est utilisé pour envoyer les données sur le réseau à les recevoir. Cela évite de bloquer l'application jusqu'à la fin de l'envoi.

BroadcastReceiver Un BroadcastReceiver est un composant utilisé pour écouter les messages en large diffusion sur le système. Un exemple de ce type de message est la réception d'un nouveau SMS. Lorsqu'un nouveau SMS est reçu par le téléphone, le système envoie un message en *broadcast* pour notifier les différentes applications d'envoi et réception de SMS. Ce composant ne possède aucune interface graphique et n'est pas censé exécuter de longues tâches.

2.1.2.2 Intents : communication entre composants

Un `intent` est un message utilisé par les composants des applications pour communiquer entre eux. Plus précisément, il est utilisé pour exécuter que le destinataire exécute une requête à la demande de l'émetteur. Le composant émettant l'`intent` et celui la recevant ne font pas forcément partie de la même application. Un `intent` a deux usages principaux : lancer un composant `Activity` ou `Service` et diffuser un message dans le système aux composants `BroadcastReceiver`. Nous parlerons alors dans ce cas de `broadcast intent`. Il existe différentes méthodes fournies par l'API Android lancer un composant `Activity` ou `Service` et diffuser un `intent` dans le système. Ces méthodes peuvent être classées selon le type de composant ciblé. Les méthodes utilisées pour envoyer un `intent` à un composant de type `Activity` et `Service` sont les méthodes dont le nom commencent respectivement par `startActivity` et `startService`. Quant aux méthodes utilisées pour diffuser un `intent` à des composants `BroadcastReceiver`, leur noms sont préfixés par `sendBroadcast`.

Un `intent` peut avoir plusieurs attributs dont quatre principaux : `component name`, `action`, `data` et `category`. Ces attributs servent à définir le(s) destinataire(s) de l'`intent` et les données à transmettre. `Component name` désigne le nom du composant destiné à recevoir le message. Si une valeur est associée à cet attribut, l'`intent` est dit explicite. Dans le cas contraire, il est dit implicite et il appartient au système de définir le destinataire du message. S'il existe plusieurs destinataires possibles, le système demandera à l'utilisateur de choisir le destinataire à qui le message sera envoyé. Ce mécanisme est appliqué uniquement lorsque les destinataires possibles sont de type `Activity`. S'il s'agit de `Service`, le système fera lui-même le choix. Dans le code du listing 2.1 par exemple, la fonction `sharePage` émet un `intent` afin de partager une URL sans définir le composant destinataire. À son émission, le système déterminera donc une liste de destinataires pouvant recevoir l'`intent` et proposera à l'utilisateur de choisir parmi les éléments de cette liste. Pour ce type d'action, Android proposera au moins dans la liste l'application de messagerie électronique, le presse papiers et le bluetooth. Pour les messages de type `broadcast intent`, le système enverra à tous les composants `Broadcast Receiver` éligibles. `Action` est une chaîne de

caractère et désigne l'action demandée par l'application. `Data` est une adresse pointant vers la donnée à traiter ou le type de données à traiter. `Category` désigne une ou plusieurs catégories de composants censés recevoir le message. Les catégories n'ont aucun lien avec les quatre types de composant d'une application Android. À partir de ces attributs, le système définit le ou les destinataires de l'`intent`.

Deux autres attributs peuvent s'ajouter aux quatre précédents : les `flags` et les `extras`. Le système ne les utilise pas pour définir le ou les destinaire(s) de l'`intent`. Les `flags` sont des attributs destinés plus au système qu'au destinataire du message. Une analyse rapide de la liste des `flags` fournis par Android montre qu'ils ont un usage assez varié. Ils servent par exemple à déléguer une permission d'accès en lecture ou écriture aux données associées à l'`intent` ou à choisir des groupes d'applications pouvant recevoir le message. Nous ne donnerons pas plus de détail sur cet attribut dans cette thèse mais la liste des `flags` fournis par Android sont disponibles dans la documentation de la méthode `setFlags` de la classe `Intent` [84]. Les `extras` permettent de transmettre des informations additionnelles au destinataire du message. Il s'agit d'une liste de couple clé valeur. Dans le code du listing 2.1, l'`intent` émis par la fonction contient quatre attributs de type `extras`. Ces attributs servent dans cet exemple à transmettre des données liées à l'URL de la page à partager : le titre de la page, une icône et une capture d'écran.

```
1  static final void sharePage(Context c, String title, String url,
2  Bitmap favicon, Bitmap screenshot) {
3      Intent send = new Intent(Intent.ACTION_SEND);
4      send.setType("text/plain");
5      send.putExtra(Intent.EXTRA_TEXT, url);
6      send.putExtra(Intent.EXTRA_SUBJECT, title);
7      send.putExtra(Browser.EXTRA_SHARE_FAVICON, favicon);
8      send.putExtra(Browser.EXTRA_SHARE_SCREENSHOT, screenshot);
9      try {
10         c.startActivity(Intent.createChooser(send, c.getString(
11             R.string.choosertitle_sharevia)));
12     } catch(android.content.ActivityNotFoundException ex) {
13         // if no app handles it, do nothing
14     }
15 }
```

Listing 2.1 – Partage d'une page web grâce à un `intent` implicite. Extrait du code du navigateur par défaut d'Android

Remarque : Afin qu'un composant A puisse recevoir un `intent` d'un composant appartenant à une application autre que la sienne, il faut que ce composant A soit déclaré avec l'attribut `exported` dans le fichier `AndroidManifest.xml` de

son application (section 2.1.2.4). Cet attribut est cependant implicite lorsqu'un `intent filter` est déclaré pour ce composant.

2.1.2.3 Intent-filter

Dans la section précédente, nous avons présenté les `intents`, un type de message utilisé par les composants d'une application pour communiquer avec d'autres composants. Lors de l'émission de l'`intent`, le système calcule à partir des attributs associés à l'`intent` le(s) destinataire(s) du message. Deux cas peuvent se présenter. Dans le premier cas, l'émetteur a défini explicitement le destinataire (*explicit intent*). Le message est donc transmis directement au destinataire choisi par l'émetteur. Dans le second cas, l'émetteur n'a pas défini de destinataire et il appartient au système de le définir (*implicit intent*). Lorsque ce cas se présente, le système définit la liste des destinataires possibles grâce à un filtre déclaré par chaque application appelée `intent filter`. Ce dernier définit pour chaque composant quels sont les `intents` que le composant souhaite recevoir et est déclaré dans le fichier `AndroidManifest.xml` qui accompagne chaque application Android. Nous donnons plus de détail sur le contenu de ce fichier dans la section 2.1.2.4 et nous nous contentons ici d'expliquer la partie sur les `intent filters`. Chaque composant d'une application est déclarée dans ce fichier et chaque déclaration est accompagnée de diverses informations telles que l'`intent filter`. L'`intent filter` déclare au système les `intents` que chaque composant peut traiter. Ce filtrage est décrit par une liste d'attributs d'`intents` que les `intents` transmis aux composants doivent avoir. Dans le listing 2.2 par exemple, l'application filtre les `intents` que deux de ses composants peuvent recevoir. Le premier composant, `App`, ne peut recevoir que les `intents` dont les attributs `action` et `category` ont respectivement comme valeur associée `android.intent.action.MAIN` et `android.intent.category.LAUNCHER`. Quant au deuxième composant, `BillingReceiver`, il ne peut recevoir que des `intents` auquel l'attribut `action` est associé à la valeur `com.android.vending.billing.RESPONSE_CODE`.

2.1.2.4 Android Package

Android Package est la forme sous laquelle une application est proposée à l'utilisateur. Il s'agit d'une archive, communément appelée `apk`, contenant le code de l'application et les ressources qu'elle utilise. Nous présentons dans ce qui suit le contenu principal d'un `apk`.

AndroidManifest.xml

`AndroidManifest.xml` est un fichier `xml` contenant les informations liées à l'application qui sont nécessaires au système. Il est créé par le développeur de l'application. Parmi les informations qu'il contient, nous pouvons citer :

— le nom du *package* de l'application
— le composant `Activity` à lancer au lancement de l'application

— la liste des composants de l'application et les informations qui sont liés aux composants (ex : permission pour accéder à un composant `Service` sensible, les `Intents` attendus par les composants de l'application etc)
— les permissions demandées par l'application (ex : READ_SMS pour lire les SMS)
— les permissions déclarées par l'application
— les bibliothèques utilisées par l'application
— le niveau minimal du SDK Android pour que l'application puisse fonctionner

Le listing 2.2 présente le fichier `AndroidManifest.xml` d'une version d'Angry Birds Space. À la deuxième ligne, sont définis l'endroit où l'application sera installée (ici laissé au choix du système) et le nom du *package* de l'application. De la ligne 4 à la ligne 11 sont ensuite listées les permissions demandées par l'application. Ici l'application demande par exemple accès à la mémoire externe, au réseau, aux informations liées au téléphone, aux données de géolocalisation et au système de payement proposé par Google.

La ligne 13 décrit comment l'application sera présentée dans le menu des applications une fois installée : l'icône à utiliser et le nom à afficher. Le reste du fichier liste les composants de l'application et les informations qui y sont liées. L'application a ainsi quatre composants dont deux de type `Activity`, un de type `Service` et un de type `BroadcastReceiver`. Comme expliqué précédemment, filtrer les messages `Intent` à destination des composants de l'application est possible. Dans `AndroidManifest.xml`, cela se fait via les entrées `intent-filter`. Ainsi le premier `Activity` ne recevra que les `intents` dont l'attribut `action` a la valeur `android.intent.action.MAIN` et la catégorie est `android.intent.category.LAUNCHER`. Cela signifie que c'est ce composant qui sera lancé lorsque l'utilisateur cliquera sur l'icône de l'application dans le menu de son téléphone. Quant au `Broadcast Receiver`, il ne réagira qu'aux `intents` dont l'attribut action vaut `com.android.vending.billing.RESPONSE_CODE`

Classes.dex

Le code de chaque classe d'une application Java standard est stocké dans des fichiers `.class` différents. Sous Android, ce n'est pas le cas. Tout est stocké dans un seul et unique fichier qui est `classes.dex`. De plus, si le code des applications Java est lui compilé en *bytecode Java*, celui des applications Android est lui compilé dans un autre format qui est le *bytecode dalvik*. C'est le contenu de ce fichier, ou plus précisément une version optimisée de celui-ci, qui sera interprété par la machine virtuelle Dalvik pour exécuter l'application.

Autres

Un `apk` contient d'autres entrées telles que les répertoires `META-INF, res, jni` et `lib`. Le répertoire `META-INF` contient ainsi des fichiers liés au contrôle d'intégrité de l'application et à l'identification de son développeur. Le répertoire `res` contient les ressources utilisées par l'application telles que des images,

sons, etc. Les répertoires `jni` et `lib` contiennent les bibliothèques utilisées par l'application.

Nous avons présenté dans cette section le système Android : le noyau, l'espace utilisateur en insistant sur les applications et la manière dont ils coopèrent. Dans la section qui suit, nous présentons les différents mécanismes de sécurité protégeant ce système.

```
<?xml version="1.0" encoding="utf-8"?> <manifest
    android:installLocation="auto"
    package="com.rovio.angrybirdsspace.ads"
    xmlns:android="http://schemas.android.com/apk/res/android">
    <uses-permission
        android:name="android.permission.READ_EXTERNAL_STORAGE"
        /> <uses-permission
        android:name="android.permission.INTERNET" />
    <uses-permission
        android:name="android.permission.ACCESS_WIFI_STATE"
        /> <uses-permission
        android:name="android.permission.ACCESS_COARSE_LOCATION"
        /> <uses-permission
        android:name="android.permission.ACCESS_NETWORK_STATE"
        /> <uses-permission
        android:name="android.permission.READ_PHONE_STATE"
        /> <uses-permission
        android:name="android.permission.WRITE_EXTERNAL_STORAGE"
        /> <uses-permission
        android:name="com.android.vending.BILLING" />

    <application android:label="Angry Birds"
        android:icon="@drawable/icon"> <activity
            android:name="com.rovio.fusion.App"
            android:launchMode="singleTask"
            android:screenOrientation="landscape">
            <intent-filter> <action
                    android:name="android.intent.action.MAIN"
                    /> <category
                    android:name="android.intent.category.LAUNCHER"
                    /> </intent-filter> </activity>

        <service
            android:name="com.rovio.fusion.GooglePlayInAppBilling.
            BillingService"/>

        <receiver
            android:name="com.rovio.fusion.GooglePlayInAppBilling.
            BillingReceiver">
            <intent-filter> <action android:name="com.android.
                    vending.billing.RESPONSE_CODE"/>
            </intent-filter> </receiver>

        <activity
            android:name="com.rovio.fusion.MediaPlayerWrapper"
            android:screenOrientation="behind"
            android:configChanges="keyboardHidden|orientation|screenSize"
            /> </application> <uses-feature
        android:glEsVersion="0x20000"
        android:required="true" /> </manifest>
```

Listing 2.2 – Fichier `AndroidManifest.xml` d'Angry Birds Space

2.2 Sécurité du système Android

Nous avons écrit dans la section précédente que le noyau d'Android était un noyau Linux avec un ensemble de modifications et d'ajouts. Une partie des mécanismes de sécurité offerts par Android proviennent ainsi du système Linux auxquels s'ajoutent des mécanismes propres à Android. Ces mécanismes ont pour but de protéger les applications les unes des autres, les communications entre applications et les ressources sensibles disponibles dans le système.

2.2.1 Mécanismes issus de Linux

Le noyau d'Android est une version modifiée du noyau Linux. Il bénéficie ainsi des mécanismes offerts par les noyaux et systèmes Linux : système multi-utilisateur, contrôle d'accès (lecture, écriture et exécution) basé sur les utilisateurs et isolation par processus.

2.2.1.1 Système multi-utilisateur et contrôle d'accès

Android supporte l'existence de plusieurs utilisateurs dans le système et utilise le mécanisme de contrôle d'accès fourni par Linux. L'accès aux différentes ressources dans le système est ainsi défini par des droits d'accès en lecture, écriture et exécution. Ces accès sont définis pour trois entités : le propriétaire, le groupe propriétaire et les autres.

Il existe un ensemble prédéfini d'utilisateurs par défaut sur les systèmes Android dont une partie est associée au fonctionnement interne du système. Parmi ces utilisateurs nous pouvons citer `root` qui est l'utilisateur avec les droits les plus élevés dans les systèmes de type Linux et Android et l'utilisateur `system` qui est associé aux différentes ressources nécessaires au fonctionnement du système tels que les bibliothèques natives partagées. Tout au long de l'exécution du système, la liste des utilisateurs peut ensuite évoluer. En effet, d'autres utilisateurs sont créés à chaque fois qu'une application est installée sur le système [2]. Ces utilisateurs ont généralement des droits restreints à savoir qu'ils ont uniquement accès aux ressources appartenant à l'application à laquelle ils sont associés. Cette restriction évite ainsi que les données de fichier appartenant à une application ne soit lues ou modifiées par une autre application.

2.2.1.2 Isolation des applications

En plus d'associer des utilisateurs différents à chaque application, Android les cloisonne également en les exécutant dans des processus différents et en leur attribuant des répertoires différents dans lesquels les applications stockent les données qu'il manipulent. Les processus associés à chaque application s'exécutent avec les droits de l'utilisateur associé à l'application et les répertoires

2. Il existe une exception à cette règle où une application peut demander à partager le même identifiant d'utilisateur que d'autres applications. Voir la section 2.1.2.4.

Type	Description
normal	Valeur par défaut des permissions. Elle est automatiquement accordée a toute application la demandant.
dangerous	Nécessite une validation de la part de l'utilisateur afin d'accorder la permission. Exemple : READ_SMS pour l'accès aux SMS.
signature	Permission accordée uniquement si l'application la demandant est signée avec le certificat du développeur ayant déclarée la permission.
signatureOrSystem	Permission accordée uniquement aux applications `system`, plus précisément celles dans la partition `system`, ou à celles ayant été signées avec le même certificat que l'application ayant déclarée la permission.

TABLE 2.1 – Types de permission Android

appartiennent également à l'utilisateur auquel l'application est associée. Ce cloisonnement évite ainsi qu'une application interfère avec l'exécution d'une autre application ou qu'elle modifie les données d'une autre application.

2.2.1.3 Chiffrement de la partition de données

Depuis la version 3.0 d'Android, le système offre la possibilité de chiffrer la partition `data`. Cette partition contient l'ensemble des applications du téléphone, à l'exception des applications fournies par défaut que l'utilisateur ne peut désinstaller, ainsi que les données qu'elles manipulent. Le chiffrement est réalisé grâce au module `dm-crypt` [15] fourni par le noyau.

2.2.2 Mécanismes propres à Android

2.2.2.1 Permissions

Android propose un ensemble de ressources sensibles aux applications installées sur le téléphone (réseau, caméra, système de géolocalisation, bluetooth, etc.). Pour les utiliser, une application devra déclarer les permissions correspondantes. Une application souhaitant lire les SMS devra par exemple avoir la permission READ_SMS. À l'installation, l'utilisateur valide les permissions demandées par l'application. Pour que l'installation se fasse, il doit toutes les accepter. Une fois l'application installée, plus aucune validation n'est nécessaire de la part de l'utilisateur. La seule exception est l'envoi de messages à des numéros surtaxés où l'utilisateur doit valider [3] chaque envoi. Il existe quatre types de permission que nous résumons dans le tableau 2.1.

3. fonctionnalité disponible depuis Android 4.2

```
Name: AndroidManifest.xml
SHA1-Digest: 9FiHXTmeVecbFb3enszaSlXIZp0=

Name: res/layout/hello_activity.xml
SHA1-Digest: BJz/aHbKT/Or0LwKJZ/jxN+WzmE=

Name: resources.arsc
SHA1-Digest: khWw+6lJ8dfajIaKkvbCuQ7YYzI=

Name: classes.dex
SHA1-Digest: +aR5lPSRcAvOrX+OhsxELXB1qGg=
```

FIGURE 2.2 – Fichier `MANIFEST.MF` de l'application HelloActivity

2.2.2.2 Signature des applications

Android requiert que chaque application soit signée afin d'être installée sur le téléphone. Les développeurs signent ainsi leur application avec un certificat dont la clé privée leur est propre.

Les applications Android sont signées avec l'outil `jarsigner`. Il prend en entrée une archive `jar` ou `zip` et un certificat. Il donne en sortie l'archive à laquelle ont été ajoutés deux fichiers : un fichier avec l'extension .SF et un autre fichier avec l'extension .RSA ou .DSA selon les clés utilisées. L'entête du premier fichier correspond au hash du fichier `MANIFEST.MF`. Le reste de son contenu est similaire au fichier META-INF/MANIFEST.MF qui est présent dans chaque archive `jar`. `MANIFEST.MF` recense pour chaque fichier présent dans l'archive son nom, son hash et l'algorithme de hachage utilisé. La figure 2.2 est le fichier `manifest.mf` de l'application `HelloActivity` fourni en exemple avec le kit de développement Android. Pour chaque fichier présent dans l'archive nous avons son nom, un algorithme de hachage et le hash du fichier. Le fichier .SF contient les mêmes types d'information que `MANIFEST.MF`. Pour chaque fichier présent dans l'archive nous avons une entrée constituée du nom du fichier, un algorithme de hachage et un hash. Contrairement à un hash dans `MANIFEST.MF`, il s'agit cette fois-ci du hash de l'entrée correspondant au fichier dans `MANIFEST.MF`. La figure 2.3 liste le contenu du fichier CERT.SF de l'application HelloActivity. Le fichier .RSA lui contient la signature du fichier .SF ainsi que la clé publique correspondant à la clé privée utilisée pour la signature.

La signature des applications Android a plusieurs objectifs. Elle sert à filtrer les applications qui peuvent être installés sur le téléphone. Par défaut, seules les applications provenant de Google Play peuvent être installées sur le téléphone. Il s'agit plus précisément des applications dont le développeur a un certificat reconnu sur Google Play. Toute application avec une signature inconnue sera bloquée à l'installation à moins que l'utilisateur n'autorise explicitement l'installation d'applications provenant d'autres plateformes.

La signature identifie également toutes les applications d'un même développeur. Si celui-ci est malveillant, il est ainsi possible d'enlever ses applications de

```
Signature-Version: 1.0
Created-By: 1.0 (Android SignApk)
SHA1-Digest-Manifest: Qenbz+ZjLsHBpHWbAHMYhLpfies=

Name: AndroidManifest.xml
SHA1-Digest: uziqi6KmjjgyRnooQ7j5ZHIKVTw=

Name: res/layout/hello_activity.xml
SHA1-Digest: who+PyjjYjRHN6maNog494Cr+CE=

Name: resources.arsc
SHA1-Digest: WlDZOLWZ+zrAmxUTKZuz99hUoZo=

Name: classes.dex
SHA1-Digest: QjwcQAkf4iVckku4qf7kiLRSndo=
```

FIGURE 2.3 – Fichier `.SF` de l'application HelloActivity

Google Play et des téléphones des utilisateurs.

La signature assure également que l'application n'a pas été modifiée par une tierce personne avant d'être publiée. La vérification se fait à l'installation de l'application. Le système vérifie que le fichier `.SF` a bien été généré par le développeur. Il vérifie ensuite que le fichier `MANIFEST.MF` n'a pas été modifié en comparant son empreinte avec celui listé dans le fichier `.SF`. Il vérifie enfin l'intégrité des fichiers dans l'archive en comparant leur empreite avec les valeurs listées dans le fichier `MANIFEST.MF`. L'installation est arrêtée si une incohérence est détectée.

Par défaut, les applications Android tournent avec des UIDs différents dans des processus distincts. Un développeur peut cependant vouloir faire tourner ses applications avec le même UID ou dans un seul processus. Il suffit pour cela qu'il le déclare dans le fichier `AndroidManifest.xml` de son application. Plus précisément, le développeur déclare un alias d'UID qui sera partagé s'il souhaite utiliser le même UID pour plusieurs de ses applications. S'il souhaite faire tourner ses applications au sein d'un même processus, il déclare un alias de nom de processus.

Pour éviter que des développeurs ne s'attribuent également le même UID ou ne s'attache au processus faisant tourner les applications d'un autre développeur, le système vérifie que l'application demandant à partager le même UID ou à être exécuté dans le processus qu'une autre application soit signée par le même développeur.

2.2.2.3 Analyse des applications

Google analyse régulièrement les applications proposées en téléchargement sur Google Play ainsi que celles en instance d'être installées sur les téléphones des utilisateurs. Si aucune information sur la nature exacte des analyses n'est fournie

par Google, nous savons qu'une analyse statique et dynamique des applications sont faites pour détecter des motifs synonymes de menace dans l'application ou des comportements malveillants. J. Oberheide et C. Miller ont montré dans [78] que les applications soumises sur Google Play sont exécutées dans une machive virtuelle et qu'il était possible de construire un profil de cette machine.

2.2.2.4 Protection de l'appareil et de ses données

Android possède un ensemble d'outils destinés aux développeurs du système. Parmi ces outils nous pouvons citer `adb` et `fastboot`. La commande `adb` permet de communiquer avec un émulateur Android ou un téléphone. Parmi les fonctions proposées, il y a l'ouverture d'un `shell` distant sur le téléphone, l'installation ou la suppression d'application et le transfert de fichier. Pour protéger le téléphone de tout usage malveillant de ces outils, la communication est possible uniquement si le mode *debug* est activé. Une authentification par clé de l'ordinateur auprès du téléphone s'ajoute également à cela depuis la version 4.2 d'Android afin de filtrer les machines pouvant communiquer avec le téléphone via `adb`. La commande `fastboot` sert à effacer/remplacer le contenu des différentes partitions sur le téléphone. Afin de l'utiliser, l'utilisateur doit dans un premier temps débloquer cette fonctionnalité au démarrage et ce processus de déblocage implique la suppression du contenu de la partition `data` du téléphone. Cette partition contient les données de l'utilisateur ainsi que des différentes applications. L'action de remplacer le contenu d'une partition est souvent désigner par l'expression *flasher* une image, l'image étant le nouveau contenu de la partition. Un attaquant désirant ainsi remplacer une partie des composants logiciels du système via `fastboot` ne pourra ainsi accéder aux données de l'utilisateur.

2.2.2.5 Administration de l'appareil

Android propose depuis sa version 2.2 une API permettant de développer des applications afin d'administrer les téléphones [83]. L'API permet de renforcer la politique sur les mots de passe (ex : taille, expiration et nombre de tentatives), imposer le chiffrement des partitions, activer / désactiver la caméra (fonctionnalité disponible depuis Android 4.0), demander la création d'un nouveau mot de passe, verrouiller le téléphone et remettre le téléphone à la configuration d'usine.

Nous avons présenté dans cette section les mécanismes de sécurité d'Android. Ces mécanismes ont pour but de protéger les applications, les ressources qu'elles utilisent et les communications entre ces processus. Ces mécanismes ne sont cependant pas parfaits et nous montrons dans la section qui suit leur limites ainsi qu'un aperçu des menaces que représentent les malware Android.

2.3 Limites des mécanismes de sécurité Android

2.3.1 Abus de permission

Les permissions donnent accès aux ressources sensibles du téléphone aux applications. Si l'utilisateur souhaite installer une application, il doit lui accorder toutes les permissions qu'elle a demandées. Si les permissions filtrent l'accès aux ressources sensibles, il n'existe cependant aucune vérification au niveau de l'usage de ces ressources. Seule la confiance aux développeurs de l'application permet de s'assurer qu'il n'y aura aucun abus. Les attaques les plus simples utilisent ainsi les permissions de manière abusive et c'est le cas de la plupart des malware ayant pour but de faire fuir des données sensibles du téléphone. Un exemple récent est une application ayant été détectée comme un logiciel espion [4] qui cible des manifestants à Hong Kong [26]. L'application demande un ensemble assez large de permissions pour espionner les utilisateurs des téléphones sur lesquels l'application est installée. Les permissions demandées donnent accès aux SMS, aux appels, à la localisation de l'utilisateur, au micro pour enregistrer l'utilisateur, etc.

Une application avec trop de permissions peut paraître suspecte aux yeux des utilisateurs avertis. Afin de ne pas éveiller la suspicion des utilisateurs, une solution pour les développeurs de malware consiste à utiliser d'autres applications présentes sur le système pour mener l'attaque ou à diviser l'attaque entre plusieurs applications qui collaboreront pour exécuter l'attaque.

2.3.2 Permissions : attaques par délégation et attaques par collusion

Une attaque par délégation [49] consiste à déléguer l'exécution de la tâche nécessitant une permission que l'application malveillante ne possède pas à une autre application qui elle la possède. Par exemple, une application n'ayant pas la permission de communiquer sur le réseau pourrait se servir du navigateur pour poster des informations ou télécharger des fichiers. Les échantillons de BadNews [91] font par exemple appel au navigateur du téléphone afin de lancer le téléchargement d'applications sur le téléphone. Une attaque par collusion consiste en une coopération entre plusieurs applications pour mener une attaque. Il n'existe aucun malware utilisant ce type d'attaque à notre connaissance. Cependant, J. Boutet et T. Leclerc ont montré la faisabilité d'une telle attaque dans [28].

2.3.3 Communication entre composants via les `intents`

Les `intents` sont des messages échangés entre les composants des applications pour transmettre des requêtes. La possibilité d'envoyer des `intents` entre deux deux composants de deux applications différentes apporte une surface d'attaque supplémentaire. Dans [34], Chin et al. décrivent en se basant sur

4. Empreinte MD5 :15e5143e1c843b4836d7b6d5424fb4a5

leur analyse du fonctionnement des `întents` des scénarios d'attaques qui pourraient exploiter cette surface d'attaque afin d'espionner les échanges de message entre application, les bloquer, les modifier, élever ses privilèges et influencer sur le comportement d'une application.

Interception des messages diffusés dans le système

Les `broadcast intents` sont des messages diffusés dans tout le système. Ils peuvent ainsi avoir un ou plusieurs destinataires. L'une des vulnérabilités qu'introduit ce type de communication est la possibilité d'observer les informations diffusées dans le système et éventuellement les intercepter, bloquer ou modifier. Lorsqu'une application diffuse un `broadcast intent`, elle définit un ensemble d'attributs qui permettent au système d'identifier les composants `BroadcastReceiver` présents dans le système qui attendent ce type de message. Pour observer les messages attendus par le composant `BroadcastReceiver` d'une application, il suffit ainsi à une application malveillante de déclarer un composant du même type avec le même `intent-filter`. Pour observer les messages reçus par le composant `BillingReceiver` déclaré dans le listing 2.2, une application malveillante n'a qu'à déclarer un composant du même type en précisant que ce composant n'accepte que les `intents` avec un attribut `action` dont la valeur associée est `com.android.vending.billing.RESPONSE_CODE`.

Un `broadcast intent` peut être transmis de manière simultané à tous les destinataires ou en suivant un ordre. Dans le cas de ce dernier, l'`intent` est transmis d'un composant à l'autre dans un ordre défini par le système. Chaque composant `BroadcastReceiver` peut ainsi modifier ou bloquer les informations transmises avant que le message ne soit transmis au prochain destinataire. Si une application malveillante se trouve au milieu de la chaîne de transmission, elle peut donc modifier le contenu de l'`intent` émis et envoyer de fausses données aux composants en attente du message. Elle peut également décider de ne pas faire suivre le message et empêcher les autres composants de le recevoir.

Détournement des intents à destination des composants Activity et Service

Lorsqu'une application émet un `intent`, il peut soit définir explicitement le destinataire soit laisser le système le définir à sa place. Dans le deuxième cas, l'émetteur n'a aucune garantie sur l'identité du destinataire ce qui donne ainsi la possibilité de détourner les messages du destinataire légitime. Une application malveillante souhaitant intercepter un `implicit intent` n'a donc qu'à déclarer un composant ayant un `intent filter` correspondant à l'`intent` qu'il souhaite détourner. Par exemple, une application malveillante souhaitant détourner le partage de page effectuée dans le listing 2.1 n'a qu'à déclarer un composant `Activity` avec un `intent filter` correspondant aux attributs de l'`intent` à intercepter : un attribut `action` et un attribut `data` auxquels sont associés respectivement les valeurs `ACTION_SEND` et `text/plain`.

Si le détournement est théoriquement possible, il n'a en réalité qu'une probabilité de succès. Lorsqu'il existe plusieurs destinataires possibles de l'`intent`, un choix qui est indépendant de l'application malveillante est effectué. Si l'`intent` a été émis pour un composant de type `Activity`, le système demande à l'utilisateur de choisir le destinataire parmi la liste des applications pouvant recevoir l'`intent`. Si l'`intent` a été émis pour un composant de type `Service`, le système effectuera lui-même le choix et ce choix est effectué de manière aléatoire.

Vol/abus des permissions

Nous avons écrit en section 2.1.2.2 que les `intents` pouvaient également servir à transmettre des permissions pour accéder à des données au destinataire du message. Positionner le `flasg FLAG_GRANT_WRITE_URI_PERMISSION` donne par exemple l'accès en lecture aux données liées à l'`intent` au destinataire du message. Une application malveillante interceptant des `intents` transmettant des permissions peut ainsi abuser de ces permissions et voler ou modifier les données auxquelles les permissions donnent accès.

Intents malveillants

Le but de cette attaque est d'envoyer des requêtes malveillantes à traiter par un composant cible. Un composant qui peut recevoir des `intents` d'autres applications n'est pas seulement exposé aux applications que son développeur pensait servir mais à toutes applications sur le système. Cette exposition à toutes les applications du système offre ainsi une surface d'attaque aux applications malveillantes qui elles aussi peut demander au composant de traiter une requête, même si cette requête est malicieuse. Les navigateurs web sous Android ont par exemple un composant `Activity` qui ouvre les URL à la demande d'autres applications. Cette fonctionnalité peut ainsi être détournée par une application qui n'a pas accès au réseau afin de faire fuir des données ou télécharger des fichiers. Pour cela l'application malveillante émettra un `intent` à destination du navigateur afin que ce dernier ouvre une adresse web. Les échantillons de BadNews [91] utilisent par exemple cette approche afin de télécharger des applications sur le téléphone.

2.3.4 Failles logicielles : élévation de privilège

Comme tout programme, le système Android a également des failles logicielles. Exploiter certaines d'entre elles permet d'élever les privilèges d'une application et ainsi exécuter des opérations sensibles que nous ne pouvions faire. Obtenir les droits `root` permet par exemple de modifier le contenu de la partition `system` sous Android pour installer des applications système ou les remplacer.

Le noyau Android étant basé sur un noyau Linux, il hérite ainsi de ses vulnérabilités. Certaines d'entre elles [5, 4] ont par exemple été exploitées obtenir des accès `root` sur les téléphones.

Des failles permettant d'élever les privilèges des applications existent également dans l'espace utilisateur. D'après les travaux de Y. Zhou et X. Jiang [113], six vulnérabilités permettant d'élever les privilèges des applications existaient au moment de leur analyse (voir tableau 2.2) et quatre d'entre elles étaient effectivement utilisées par les malware pour élever leurs privilèges : Asroot [1], exploid [31], RATC / Zimperlich [93] et GingerBreak [32]. Plus récemment, J. Forristal a présenté à la Black Hat 2013 une vulnérabilité [51] concernant la vérification des signatures des applications Android à l'installation. La vulnérabilité permet d'installer une version modifiée d'une application dont la signature reste celle de la version originale. Si la vulnérabilité ne permet pas d'obtenir les droits `root` (aucune application ne tourne avec l'UID `root`), elle rend cependant caduque les protections offertes par la signature en section 2.2.2. Un développeur malveillant peut faire exécuter son code avec les mêmes droits que l'application originale qu'il a modifiée. Il peut également faire tourner son code dans le même processus ou avec le même UID qu'une autre application. S'il n'est pas possible d'obtenir les droits `root`, il est cependant possible d'obtenir les droits des applications `system`. Ces applications ont accès à plus de permissions que les applications tierces et de plus sont persistantes sur le système. Un utilisateur ne peut les enlever sans avoir un accès `root` sur son téléphone.

Bien que Google mette à jour régulièrement le code d'Android, les constructeurs eux mettent plus de temps à proposer des mises à jour pour leur téléphone. La fenêtre d'exploitation des vulnérabilités est ainsi bien plus large que sur les ordinateurs.

Nous avons présenté dans cette section, les limitations des mécanismes de sécurité sous Android. Ces limitations peuvent être classées en trois groupes. Le premier groupe concerne les limites de la sécurité offerte par les permissions. Le second concerne les risques introduits par les communications entre composants via les `intents`. Le troisième groupe concerne les failles logicielles dans le code d'Android qui permettent d'élever les privilèges des applications dans le système. Dans la section suivante, nous présentons les malware Android et les menaces qu'ils représentent.

2.4 Malware Android

2.4.1 Définitions

Nous appelons malware un programme ou un code dont le but est de nuire à un système donné. Dans le reste du document nous ferons souvent usage des termes échantillon de malware et famille de malware. Un échantillon d'un malware est une application, comprendre ici application Android correspondant à un fichier `apk`, qui contient ce malware. Quant à une famille de malware, il s'agit de l'ensemble des échantillons d'un malware. Analyser un malware revient ainsi

à analyser un ou plusieurs de ses échantillons afin d'extraire des informations liées au malware et détecter un malware revient à décider si une application donnée est un échantillon d'un malware.

Les premiers travaux qui ont consisté à dresser un bilan des menaces que représentent les malwares Android sont les travaux de Y. Zhou et X. Jiang dans [113]. Leurs travaux sont basés sur l'analyse de plus de 1200 échantillons de malware qu'ils ont collecté de 2010 à 2011. Dans ce qui suit, nous présentons les résultats de cette analyse et l'enrichissons avec une analyse plus récente basée sur les menaces que représentent les malware Android en 2013.

2.4.2 Malwares Android : 2010 à 2011 [113]

Méthode d'infection

Pour infecter les téléphones des utilisateurs, les développeurs de malware ajoutent leur code malveillant à des applications existantes et proposent leur version modifiée sur des plateformes de téléchargement : Google Play ou toute autre plateforme alternative. Les applications infectées sont présentées comme une version gratuite d'une application payante ou des versions avec plus de fonctionnalités. 86% des échantillons récoltés sont ainsi des applications originales auxquelles un code malveillant a été ajouté. Le type d'application infectée est varié : application payante, jeux populaires, outils tels que des mises à jour de sécurité ainsi que des applications pour adultes. Un des échantillons de DroidKungFu2 est par exemple une version modifiée d'une application simulant un scanner d'empreinte pour déverrouiller le téléphone. En arrière plan, le code malveillant exploite une vulnérabilité [92, 93] pour élever ses privilèges et installer des binaires sur le téléphone à l'insu de l'utilisateur.

Tout le code malveillant ne se trouve pas forcément ajouté à l'application originale. Certains échantillons ne contiennent ainsi qu'une partie du code malveillant dont le reste sera récupéré à l'exécution. Ce dernier peut être stocké en tant que ressource de l'application ou sur un serveur distant que l'application infectée devra télécharger. L'avantage étant que le code malveillant ne pourra être détecté lors de la soumission de l'application sur les plateformes de téléchargement. Les échantillons d'AnserverBot [112] ont ainsi une charge cachée en tant qu'image dans les ressources des applications et récupèrent également une autre application malveillante sur un serveur distant.

Déclenchement du code malveillant

Le déclenchement du code malveillant ne se fait pas forcément dès le lancement de l'application. Le code malveillant peut attendre des évènements spécifiques tels que l'arrivée d'un SMS ou l'écoulement d'un certain laps de temps avant de s'exécuter. Parmi les évènements les plus utilisés pour déclencher le

Programme vulnérable	Exploit	Date	Malware les utilisant
Noyau	Asroot	2009/08/16	Asroot
init ($\leqslant$ 2.2)	Exploid	2010/07/15	DroidDream, zHash, DroidKungFu*
adbd ($\leqslant$ 2.2.1) zygote ($\leqslant$ 2.2.1)	RATC / Zimperlich	2010/08/21 2011/02/24	DroidDream, BaseBridge, DroidKungFu, DroidDeluxe, DroidCoupon
ashmem ($\leqslant$ 2.2.1)	KillingInTheNameOf	2011/01/06	Aucun
vold ($\leqslant$ 2.3.3)	GingerBreak	2011/04/21	GingerMaster
libsysutils ($\leqslant$ 2.3.6)	zergRush	2011/10/10	Aucun

TABLE 2.2 – Exploits `root` connus et leur usage par les malware Android de 2010 à 2011

code malveillant, nous pouvons citer la fin d'initialisation au démarrage du système. Quand le système est prêt à exécuter des applications Android, il le signale en envoyant un `intent` avec le message BOOT_COMPLETED à tous les composants BroadcastReceiver à l'écoute cet évènement. Les autres messages évènements servant de déclencheur sont la réception de message, les appels, les évènements liés aux applications (ex : ajout, suppression, mise à jour et redémarrage), l'état de la batterie, l'état de la connectivité réseau, et des évènements systèmes (ex : carte SIM remplie, changement de clavier). Le développeur du code malveillant ajoute un composant de type `BroadcastReceiver` pour intercepter l'un de ces évènements et exécuter par la suite le code malveillant.

Actions effectuées

Les actions effectuées par les codes malveillants peuvent être réparties dans quatre groupes : élévation de privilège, contrôle à distance, charge financière et vol de données.

Comme évoqué en section 2.2.2, Android possède des vulnérabilités qui sont exploitées par les développeurs malveillants pour effectuer des opérations sensibles. Le tableau 2.2 montre d'ailleurs une tendance à utiliser les exploits `exploid` [31], `RageAgainstTheCage` [92] et `Zimperlich` [93] afin élever les privilèges des applications malveillantes durant l'attaque. Le but de cette élévation de privilèges est d'effectuer des opérations sensibles telles que monter la partition `system` avec l'option d'écriture pour y installer de nouvelles applications.

1172 échantillons soit 93% des échantillons analysés sont contrôlables à distances. Plus précisément, 1171 échantillons utilisent le protocole `HTTP` pour re-

cevoir des commandes des serveurs de commande et contrôle C&C. Les adresses des serveurs sont stockées en clair ou chiffrées dans le code.

En dehors des élévations de privilèges et communication avec des serveurs de C&C, les échantillons analysés peuvent imputer des charges financières à l'utilisateur. Android tournant principalement sur les *smartphones*, les applications ont ainsi accès aux fonctions d'appels et de SMS. 4.4% des échantillons étudiés envoient ainsi des messages à des numéros surtaxés. Les numéros sont soit stockés en dur dans le code de l'application soit récupérés à partir des serveurs de C&C.

En plus des actions précédentes, les malware récupèrent également les données sensibles sur le téléphone. Les données ciblées sont principalement les SMS, la liste de contact, et les informations sur le compte de l'utilisateur. SndApps [62] collecte par exemple les adresses courriel de l'utilisateur et les envoie vers un serveur distant.

Évolution des malware

Si les premiers malware Android sont simples, l'analyse effectuée dans [113] montre que les techniques utilisées tendent à se complexifier : charge utile, techniques contre l'analyse d'application, serveurs C&C.

Tout le code malveillant est intégré dans une même application dans les premiers malware. L'analyse des échantillons récoltés montre une tendance à le diviser, mettre une partie dans l'application servant à l'infection et le reste des morceaux dans une ou plusieurs charges utiles. Ces charges sont intégrées directement dans l'application ayant servi à infecter le téléphone ou télécharger et après l'installation de l'application. Les échantillons de DroidKungFu ont ainsi deux charges utiles cachées en tant que ressources des applications infectées qui seront installées dans la partition `system` une fois l'échantillon lancé. L'installation de ces charges sur le téléphone offre une présence constante sur le téléphone même si la première application est enlevée du téléphone. De plus, si les applications sont installées sur la partition système, il n'existe aucun moyen pour l'utilisateur de les enlever sans avoir les droits `root` sur le téléphone.

Afin d'éviter toute détection, les développeurs de malware utilisent diverses techniques telles que le chiffrement des charges utiles, le chiffrement de certaines valeurs utilisées par le malware, l'usage de techniques d'obfuscation de code et l'usage de code natif. Le chiffrement permet de cacher la nature exacte des données manipulées (ex : adresses des serveur de C&C) par l'application et réduit ainsi la possibilité de détection. Les développeurs de DroidKungFu1 chiffrent ainsi les charges utiles de leur malware afin que leur malware ne soit détecté facilement. Si une clé de chiffrement différente est utilisée par chaque échantillon, l'analyse sera encore plus compliquée. Certains développeurs de malware obfusquent également leur code afin de complexifier son analyse [39]. Ils modifient par exemple le nom des méthodes et des classes Java de l'application. Il est cependant à noter que le kit de développement Android (Android SDK) contient un outil fournissant le même type de service [87]. L'usage de code natif complexifie également l'analyse des applications. En effet, la plupart des outils

d'analyse d'application Android se concentrent uniquement sur le code écrit en Java car c'est le langage principal pour développer une application Android. En utilisant du code natif, les développeurs de malware se donnent ainsi la possibilité de cacher une partie du code malveillant durant l'analyse.

Certains malware intègrent également des modules censés prévenir toute détection. Les échantillons d'AnserveBot analysent ainsi son environnement d'exécution à la recherche d'antivirus pour Android. De plus, ces échantillons tirent également profit du chargement dynamique de code [36]. Le code des applications Android se trouve dans le fichier `classes.dex` de leur `apk`. Ce mécanisme permet de charger du code en dehors de ce fichier et ce de manière dynamique rendant l'analyse de l'application plus difficile. D'autres malwares vérifient également que le code n'a pas été modifié pour détecter toute tentative d'analyse du code. AnserveBot vérifie par exemple l'intégrité de son code avant de lancer le code malveillant.

Enfin, l'analyse montre également l'usage des serveurs de C&C afin de contrôler le comportement des échantillons de malware. Le comportement exact des premiers malware sont dictés à l'avance par leur développeur. Durant leur analyse, Y Zhou et X Jiang ont cependant constaté que dans le cas de certains malware, leur comportement était plutôt dicté par un serveur distant. Le malware se connectait ainsi périodiquement au serveur qui lui envoyait par la suite l'action qu'il devait exécuter. Sur les 49 malwares étudiés, 27 utilisent ainsi un serveur de C&C pour recevoir les commandes à exécuter sur le téléphone infecté. L'usage de telles approches permet aux développeurs de garder une flexibilité sur les différentes actions à exécuter et les faire évoluer en cas de besoin.

2.4.3 Malwares Android en 2013

Si les travaux de [113] présentés précédemment concernent uniquement les malwares de 2010 à 2011, la récente analye de V. Chebyshev et R. Unuchek dans [33] montre une continuité dans les types d'action menée par les malware.

La distribution des malwares se fait toujours via les plateformes de téléchargement ou les serveurs de C&C. À cela s'ajoute, l'usage des techniques telles que le *drive-by download* qui consiste à faire télécharger automatiquement puis installer une application au téléphone lorsque l'utilisateur visite une page web.

L'usage de techniques contre les protections anti-malware s'intensifie également. L'étude montre ainsi que les développeurs de malware continuent leur investissement dans les diverses techniques d'obfuscation de code. Un outil commercial d'obfuscation de code aurait été par exemple utilisé sur Opfak.bo et Obad.a [98]. Obad.a est considéré comme le malware Android le plus complexe à ce jour. Parmi les caractéristiques de ce malware, nous pouvons citer l'introspection, le chiffrement de chaine des chaines de caractères, l'exploitation de vulnérabilités qui affectent le système Android et `dex2jar` qui est un outil utilisé pour analyser les applications Android. Plus précisément, cet outil transforme le *bytecode dalvik* en *bytecode Java* permettant par la suite d'obtenir un code Java "équivalent". Selon *wikibooks* [12], "la réflexion permet l'introspection des classes, c'est-à-dire de charger une classe, d'en créer une instance et d'accéder

aux membres statiques ou non (appel de méthodes, lire et écrire les attributs) sans connaître la classe par avance“. Le listing 2.3 présente un bout code en Java utilisant la réflexion et son équivalent sans réflexion. . La deuxième ligne initialise la variable `foo` en une instance de la classe `Foo` et est équivalente à la septième ligne. La troisième et la quatrième lignes initialisent récupèrent la méthode `hello` de `foo` et l'invoque. Elles correspondent à la huitième ligne.

```
1  // Avec reflexion
2  Object foo = Class.forName("complete.classpath.and.Foo").
3          newInstance();
4  Method m = foo.getClass().getDeclaredMethod("hello",
5          new Class<?>[0]);
6  m.invoke(foo);
7
8  // Sans reflexion
9  Foo foo = new Foo();
10 foo.hello();
```

Listing 2.3 – Exempe de code Java avec et sans réflexion

L'usage des vulnérabilités reste toujours d'actualité au niveau des malware. Les raisons principales de cet usage est la nécessité d'effectuer des actions sensibles sans validation de l'utilisateur (ex : installation d'application sur le téléphone) ou le maintien d'une présence pertinente du malware sur l'appareil (ex : installation d'une application dans la partition `system` afin d'empêcher sa désinstallation [5]). Parmi les vulnérabilités émunérés par les auteurs, il y a la vulnérabilité *Master Key* [51] et celle liée aux applications avec les droits d'administration sur le téléphone.

La vulnérabilité Master Key permet de modifier une application existante sans que la modification ne soit détectée lors de la vérification de la signature de l'application (section 2.2.2.2). Pour exploiter cette vulnérabilité, le développeur malveillant ajoute de nouveaux fichiers à l'`apk` tel que chaque nouveau fichier ait le même nom qu'un fichier existant dans l'`apk` d'origine. Il peut par exemple ajouter un autre fichier nommé `classes.dex`. Une fois les nouveaux fichiers ajoutés, il publie l'`apk` modifié tout en gardant la signature de l'`apk` d'origine. À cause de la vulnérabilité, le système ne remarque pas l'existence des doublons dans l'`apk` et vérifie uniquement l'intégrité des fichiers qui étaient présents dans l'`apk` d'origine tandis qu'à l'installation il installera les fichiers ajoutés par le développeur malveillant. Cette vulnérabilité a deux conséquences directes. La première est le vol de l'identité des développeurs dont l'application a été modifiée. En gardant la signature de l'application originale, le développeur malveillant se cache derrière l'identité des développeurs des applications modifiées. Le risque pour ces derniers est d'être accusé à tord comme étant malveillant

5. Un utilisateur ne peut désinstaller les applications dans la partition `system` car cette partition est montée en lecture seule

et voir toutes leurs applications supprimées des plateformes de téléchargement tels que Google Play. La deuxième conséquence de cette vulnérabilité est la possibilité de contourner toutes les mécanismes de sécurité basées sur les signatures de applications. En plus de servir à la vérification de l'intégrité des applications à leur installation, les signatures servent également à appliquer des contraintes sur certaines demandes des applications. Ils peuvent par exemple servir à restreindre les applications à qui une permission peut être accordée (tableau 2.1) ou les applications qui ont le droit de partager le même utilisateur. En gardant la signature de l'application qu'il a modifiée, le développeur malveillant s'assure ainsi que sa version modifiée de l'application ait les mêmes privilèges que l'application d'origine.

La deuxième vulnérabilité permet de garder une présence persistante sur le téléphone grâce aux droits d'administration. Lorsqu'une application obtient les droits d'administration sur le téléphone, il devient impossible pour l'utilisateur de lui révoquer ces droits ou le désinstaller. Si jamais l'utilisateur tentait de révoquer les droits d'administration, le système se contentait de masquer le fait que l'application possède ces droits mais ne les révoquait pas. L'application reste ainsi installée sur le téléphone avec les droits d'administration sans que l'utilisateur ne soit au courant.

Si les différents types d'attaque présentés dans [113] restent toujours d'actutaité, l'analyse montre cependant une tendance pour les attaques ayant un impact financier sur l'utilisateur. Aux applications abusant des services de SMS en envoyant des messages à des numéros surtaxés s'ajoutent ainsi les applications se faisant passer pour des applications bancaires afin de voler les données bancaires des utilisateurs telles que leur numéro de compte.

2.5 Renforcement de la sécurité sous Android

2.5.1 Protection des ressources sensibles

2.5.1.1 TaintDroid

Dans [46], Enck et al. présentent TaintDroid une version modifiée d'Android capable de suivre les flux d'information dans le système. Le but de leur travail est d'étudier si les applications Android font fuir des données sensibles vers des entités distantes. Pour cela, ils sélectionnent un ensemble d'informations qu'ils jugent sensibles telles que la liste de contact et les données de géolocalisation et observent comment elles se propagent dans le système. Ils ont analysé 30 des applications les plus populaires de Google Play et ont montré que 2/3 d'entre elles faisaient fuir des informations sensibles vers des serveurs distants.

Pour suivre les flux d'information dans le système, ils utilisent une méthode dite de *tainting* qui consiste à marquer les informations sensibles afin d'en suivre la propagation. Chaque fois qu'une information sensible se propage, le conteneur destination reçoit la marque de l'information sensible pour caractériser son nouveau contenu. Dans TaintDroid, le suivi de flux d'information se fait à différents niveaux : à l'intérieur de l'application, entre les applications et entre applications

et fichiers du système. Pour suivre les flux à l'intérieur des applications, Enck et al. ont modifié la machine virtuelle Dalvik. Lorsqu'une application Android est exécutée, son code est interprété par la machine virtuelle Dalvik. TaintDroid suit ainsi les flux d'information entre les conteneurs d'information que la machine Dalvik utilisent. Ces conteneurs sont les variables locales d'une méthode, ses paramètres, les champs statiques d'une classe, les champs des instances d'une classe et les tableaux. À chaque fois qu'une instruction décrivant un flux d'information explicite entre deux ou plusieurs conteneurs est interprétée par la machine virtuelle Dalvik, TaintDroid considère que le conteneur destination du flux contient le mélange des informations provenant des conteneurs source. Par exemple, l'affectation à la variable v_s de la valeur de la variable v_d une mise à jour de la marque associée v_s. Lorsque cette affectation est interprétée par la machine virtuelle Dalvik, TaintDroid considère que v_s les informations contenues dans v_s ont été remplacées par celles dans v_d et TaintDroid affecte ainsi la marque associée à v_d à v_s.

Le suivi de flux d'information entre les applications et entre une application et un fichier est plus simple. Le suivi de flux d'information entre les applications est assez trivial. Comme il existe un ensemble de classes et de méthodes Java fourni par Android pour implémenter les mécanismes de communication, TaintDroid se contente de marquer les objets avec un contenu sensible quand ils sortent d'une application et à marquer le destinataire d'une information provenant d'une application externe à l'application. Les marques de chaque conteneur sont stockés dans une mémoire adjacente au conteneur et propagées à chaque fois que ces conteneurs sont envoyées vers d'autres application. Quant aux flux d'information entre une application et un fichier, TaintDroid utilise les attributs étendus. Les attributs étendus sont des fonctionnalités offertes par les systèmes de fichier tels que ext2/ext3/ext4 (système de fichier utilisé principalement sur Android) permettant de stocker des méta-données liées aux fichiers. TaintDroid stocke ainsi dans les attributs étendus d'un fichier les marques des informations sensibles qu'il contient. Lorsqu'une application accède à un fichier, TaintDroid met à jour la marque associée au fichier (écriture) ou à la variable recevant le contenu du fichier (lecture).

Si TaintDroid est capable de suivre les flux d'information dans une application, entre les applications Android et entre une application et un fichier, sa portée est cependant limitée aux applications écrites en Java. En effet, le mécanisme de suivi de flux dans TaintDroid repose principalement sur la machine virtuelle Dalvik. Or, il est possible d'utiliser du code natif sous Android. Cela se fait, soit en utilisant le mécanisme de JNI [86] soit en exécutant une application compilée en code natif. Lorsque ces cas se présentent, TaintDroid devient ainsi incapable de suivre les flux d'information causés par le code natif car il n'est pas interprété par la machine virtuelle Dalvik.

2.5.1.2 Contrôle d'accès aux ressources sensibles à l'exécution : MockDroid et AppFence

Sous Android, l'utilisateur valide les permissions à l'installation des applications. À l'exécution, il n'a plus aucun contrôle sur les accès et ne peut que faire confiance aux applications pour ne pas utiliser de manière malintentionnée les ressources et données auxquelles elles ont accès. Afin de palier cette limite d'Android, MockDroid [24] et AppFence [59] proposent à l'utilisateur de contrôler l'accès effectué par les applications durant leur exécution. Lorsqu'une application souhaite accéder à une donnée ou ressource sensible, le système demande une validation de la part de l'utilisateur. L'accès aux informations et ressources sensibles se font via des fonctions de l'API Android. Les auteurs de MockDroid et d'AppFence ont ainsi modifié Android afin d'intercepter les appels à ces fonctions et introduire le mécanisme de validation lors de leur appel. Selon l'information ou la ressource demandée par l'application, si l'utilisateur lui en refuse l'accès, le système soit lui notifiera son indisponibilité ou son inexistence, soit lui renverra une donnée factice. Les informations et ressources sensibles dont l'accès est renforcé sont les données de géolocalisation, la liste de contact, les informations liées à l'identité du téléphone, les SMS, l'envoi de message de type `broadcast intent` (MockDroid), les journaux d'évènement (AppFence) et les données de navigation internet (AppFence).

En cas de refus d'un accès à une information sensible, MockDroid et AppFence renvoient une donnée vide à l'application. L'exception est l'accès aux données identifiant l'appareil. En cas de refus, le système renverra une donnée factice à l'application. Dans AppFence l'exception s'étend aux données de géolocalisation où le système enverra les mêmes coordonnées factices à chaque fois. En cas de refus d'un accès réseau, le système simule l'indisponibilité du réseau. AppFence fait par exemple croire à l'application que le téléphone est en mode avion.

Si AppFence et MockDroid ont les mêmes objectifs et approches, la différence entre les deux réside dans le fait qu'AppFence propose également un meilleur mécanisme pour protéger les fuites de données vers des serveurs distant. Pour empêcher toute fuite de donnée, MockDroid simulera l'indisponibilité du réseau à toute tentative de connexion. AppFence propose une approche plus fine en demandant uniquement une validation quand une donnée sensible est susceptible de quitter le système. Pour ce faire, AppFence intègre le mécanisme de suivi de flux d'information implémenté dans TaintDroid [46]. TaintDroid est capable de suivre les flux d'information au sein d'une application, entre les applications et entre les applications et les fichiers. Ainsi AppFence ne demande la validation de l'utilisateur uniquement quand les flux observés au sein d'une application indiquent une fuite d'information vers l'extérieur. Si c'est le cas et en cas de refus de l'utilisateur, AppFence bloquera l'envoi en simulant l'indisponibilité du réseau ou omettra l'envoi des données sensibles tout en faisant croire le contraire à l'application.

2.5.1.3 Protection de contenu : Porscha

Dans [79], Ongtang et al. proposent un mécanisme de protection de contenu pour Android du nom de Porscha. Porscha permet de lier des données sensibles à un appareil et un ensemble défini d'applications. À l'émission d'un contenu sensible, sa source lui associe une politique de sécurité qui définit les destinataires du contenu et les conditions sous lesquelles le contenu peut être accédé. Une politique peut par exemple être une liste d'empreinte MD5 des applications autorisées à accéder au contenu protégé ainsi que des coordonnées GPS auxquelles le téléphone doit se trouver au moment de l'accès au contenu. Le contrôle d'accès aux données sensibles se fait à différents niveaux : à la transmission des données vers le téléphone et la transmission des données aux applications sur le téléphone. Pour protéger le contenu lors de sa transmission vers un téléphone, Ongtang et al. proposent de chiffrer le message en utilisant un système de chiffrement basé sur l'identité du destinataire [27]. Dans leur travail, l'identité est soit le numéro du téléphone soit une adresse mail. Les messages contenant un contenu protégé sont ainsi chiffrés et ne peuvent être lus que par leurs destinataires.

Sur le téléphone, Porscha vérifie l'accès aux données sensibles en ajoutant des points de contrôle dans les mécanismes de communication Android : à la transmission des données sensibles aux applications de messagerie et lors des communications entre applications. À la réception des MMS et des courriels, Porscha vérifie si une politique de sécurité est associée au contenu du message. Si c'est le cas, il transmet uniquement le contenu du message aux applications autorisées à le recevoir. Dans le cas des MMS, les applications non autorisées ne recevront pas le message. Dans le cas des courriels, le message est transmis à toutes les applications courriels mais celles qui ne sont pas autorisées à accéder aux données sensibles ne recevront que le message vidé de son contenu. Une fois les données sensibles stockées sur le téléphone, une application peut souhaiter partager les données sensibles à d'autres applications de confiance sur le système. Afin de contrôler ce partage, Porscha contrôle les communications et partages de données utilisant les `intents`, les composants de type ContentProvider et les appels de méthodes distantes. Pour protéger les données transmises via ces mécanismes, une application attache la politique de sécurité des données au message servant à transmettre les données. Lors de la transmission du message, Porscha vérifie la présence d'une politique de sécurité dans le message et si c'est le cas limite la transmission aux destinataires autorisés par la politique.

Pour résumer, le système de protection de contenu offert par Porscha permet de définir de manière plus fine à qui des données sensibles peuvent être transmises et sous quelles conditions. Cependant, contrairement à AppFence ou TaintDroid qui suivent la propagation des informations dans tout le système (applications écrites en Java et fichiers), Porscha ne résout pas le problème de l'usage de l'information. En effet, Porscha ne protège les données que lorsqu'elles partent de leur application source (application partageant les données en leur associant une politique de sécurité). Une fois que les données sont transmises à d'autres applications, il n'y a plus de contrôle sur l'usage qui est en est fait.

Le destinataire peut ainsi faire fuir les données sans que le système ne soit au courant.

2.5.1.4 AppIntent

Une fuite d'information ne signifie pas forcément qu'une attaque ait eu lieu. Selon les cas, la fuite peut être intentionnelle (ex : partage de coordonnées géographiques par messagerie), c'est-à-dire faite par l'utilisateur même, ou exécutée par l'application sans réelle intervention de l'utilisateur. Dans le cas d'une fuite intentionnelle, le risque d'une attaque est moindre. Dans le second cas, fuite des données sans que l'utilisateur n'ait initié l'action, la probabilité qu'une attaque ait lieu est élevée car l'utilisateur n'a pas conscience de la fuite.

Dans [106], Yang et al. présentent AppIntent, un outil qui analyse les applications Android afin de déterminer les évènements et entrées attendus par une application et entraînant une fuite d'information. Le but d'AppIntent est de fournir à un analyste les évènements et entrées attendus par une application menant à la fuite d'une information afin que l'analyste puisse statuer de la nature de la fuite : intentionnelle ou non (risque d'une attaque).

Pour calculer les évènements et entrées entraînant une fuite, l'outil analyse statiquement le code de l'application. Dans un premier temps, il construit son graphe de flux de contrôle puis détermine à partir de ce graphe les chemins d'exécution contenant une fuite d'information. Une fuite d'information est un ensemble d'instruction qui récupèrent une donnée sensible puis la fait fuir. Une fois ces chemins calculés, AppIntent calcule la suite d'évènements ainsi que les entrées attendues qui mènent à l'exécution de la fuite d'information. Les évènements sont les créations et exécutions des composants d'une application (ex : création et exécution d'un composant Activity ou la reprise d'exécution du composant) ainsi que les interactions avec l'interface utilisateur (ex : validation d'un formulaire) et les entrées sont les données liées à ces évènements tels que les attributs d'un `intent` utilisé pour lancer l'exécution d'un composant ou les valeurs des champs d'un formulaire.

À partir des évènements et entrées calculés, AppIntent crée ensuite des tests unitaires qui correspondent qui exécutent ces suites d'évènement et donnent à l'application les entrées menant à la fuite d'une information. Pour déterminer de la nature des fuites d'information, l'analyste n'a ainsi qu'à exécuter ces tests unitaires et statuer si lors de l'observation d'une fuite, les actions menées à cette fuite sont intentionnelles ou non.

Pour évaluer leur outil, Yang et al. ont analysé 750 échantillons de malware et 1000 applications parmi les plus populaires dans la section des applications gratuites sur Google Play. Sur les 750 échantillons malware, AppIntent a détecté 219 fuite de données non intentionnelles et 17 fuites intentionnelles (faites par l'utilisateur). Quant aux applications de GooglePlay, AppIntent a détecté 26 cas de fuites de données non intentionnelles et 29 cas de fuites intentionnelles. En comparaison, TaintDroid a détecté moins de fuites de données : 125 dans le cas des échantillons de malware et 40 pour les applications de Google Play.

2.5.2 Communication entre processus et entre composants

La délégation d'une partie de l'attaque à d'autres applications présentes sur le système et la collaboration entre applications permettent aux applications malveillantes de cacher leur véritable nature aux yeux des utilisateurs. Par exemple, une application demandant accès aux photos, aux données de géolocalisation, à la liste de contacts et à internet paraît suspecte car elle possède assez de permission pour faire fuir les données liées à l'utilisateur. Par contre, la même application avait l'accès internet en moins paraît moins suspecte car rien n'indique qu'elle pourrait communiquer avec une entité distante. Or comme nous l'avons expliqué dans la section 2.3.2, les applications peuvent utiliser d'autres applications ou collaborer avec d'autres applications pour mener une action qu'elles ne pouvaient faire toutes seules. Dans ce qui suit, nous présentons ainsi les travaux visant à bloquer ce type d'attaque sur Android en contrôlant les communications entre applications.

2.5.2.1 ComDroid

Nous avons présenté en section 2.3.3 le résultat d'une analyse sur les vulnérabilités introduites par les communications basées sur les `intents`. Pour détecter ces vulnérabilités, Chin et al. ont ainsi présenté dans [34] un outil d'analyse d'application du nom de ComDroid. ComDroid effectue deux types d'analyse sur le code des applications et leur fichier `AndroidManifest.xml`.

La première analyse consiste à analyser le code des applications. Durant cette analyse, ComDroid étudie comment les `intents` sont créés puis émis par une application. ComDroid émet une alerte à chaque fois qu'il détecte qu'un `implicit intent` est émis avec une restriction trop faible sur les destinataires. Une restriction est faible si aucun filtrage basé sur les permissions du destinataire n'est imposé ou si les permissions imposées sont de type `normal` (voir tableau 2.1). Une permission de type `normal` est une permission automatiquement accordée à toute application la demandant.

La deuxième analyse consiste à analyser les composants des applications afin de déterminer s'il y a un risque que les composants reçoivent des requêtes malveillantes via les `intents`. ComDroid considère qu'il y a une possibilité pour qu'un composant soit vulnérable si ce dernier peut recevoir des `intents` provenant d'autres applications [6] telles qu'il n'y ait aucune restriction basée sur les permissions ou une restriction trop faible sur les émetteurs de l'`intent`. Tout comme dans la première analyse, une restriction trop faible basée sur les permissions est une restriction basée uniquement sur les permissions de type `normal`.

Si l'analyse des communications via les `intents` montre les risques que ces communications introduisent, il n'existe cependant pas de méthode pour déterminer si ce qui a été détecté comme un risque de sécurité est une vulnérabilité de l'application ou une fonctionnalité implémentée par le développeur. En effet, le fait qu'un `implicit intent` soit utilisé ou qu'un composant soit exposé au reste du système peut traduire une volonté de s'adapter aux applications

6. Un attribut `exported` ou un `intent filter` est défini pour le composant

présentes dans le système. L'exemple typique est l'ouverture d'une page web à partir d'une application autre que le navigateur.

Du côté du développeur de l'application demandant l'ouverture de la page web, le choix d'envoyer un `implicit intent` au lieur de définir explicitement le navigateur permet de s'assurer que le lien soit ouvert qu'importe le navigateur sur le système. Il existe plusieurs navigateurs web sur Android (ex : Google Chrome et Firefox) et l'utilisateur a la possibilité de choisir quel navigateur installer et utiliser sur le téléphone. Définir explicitement le destinataire de la requête pour ouvrir le lien revient ainsi à courir le risque que le lien ne soit pas ouvert car le navigateur ciblé n'est pas présent sur le téléphone. Au contraire, en envoyant un `implicit intent` que tous les navigateurs peuvent recevoir, le développeur s'assure que le lien sera ouvert s'il existe un navigateur sur le téléphone.

Le même choix se présente également aux développeurs de navigateurs web. En exposant leur navigateur à toute application présente dans le système, les développeurs s'assurent que leur application soit compatible avec toute application souhaitant ouvrir une page web.

2.5.2.2 Saint

Les permissions sont les bases du filtrage des communications entre applications Android pour protéger l'accès à des ressources sensibles. Le contrôle de leur attribution et de leur usage est cependant assez faible sur Android. En effet, une application ne peut restreindre l'attribution d'une permission qu'elle a déclarée à l'installation d'une nouvelle application qu'en déclarant la permission comme étant `signature`. Dans ce cas, la permission n'est accordée que si l'application la demandant a été signée par le même développeur que celui de l'application l'ayant déclarée. Une fois l'application demandant la permission installée sur le téléphone, il n'existe également aucune restriction sur son usage. Le développeur d'une application pourrait par exemple souhaiter que les applications communiquant avec la sienne ne puisse le faire que dans des contextes précis tels qu'une plage d'horaire.

Partant de ce constat, Ongtang et al. proposent dans [80] une extension de sécurité à Android du nom de Saint dont le but est de donner plus de contrôle sur l'attribution des permissions et leur usage. L'apport de Saint consiste en un module permettant des restrictions supplémentaires en plus des permissions à l'installation des applications et à leur exécution, ou plus précisément lors des communications entre processus.

Saint permet aux applications de définir une politique de sécurité qui contrôle l'attribution des permissions protégeant leurs interfaces à l'installation. Par défaut, la seule manière de limiter l'éligibilité d'une application pour l'obtention d'une permission est de déclarer cette dernière comme étant de type *signature*[7]. Ce type de permission n'est accordée qu'aux applications signées avec la même clé que l'application ayant déclaré la permission. Grâce à Saint, les applications

7. Voir le tableau 2.1 pour la liste des types de permission

peuvent déclarer des restrictions plus fine sur l'attribution des permissions à l'installation. Elles peuvent ainsi imposer une restriction sur une liste développeurs au lieu d'un seul qui est le développeur de l'application ayant déclarée la permission, les autres permissions demandées par l'application, son numéro de version, etc. La politique d'attribution définit un comportement par défaut (attribuer ou refuser) et les critères qui y font exception. Une application peut ainsi déclarer qu'une permission n'est accordée à une autre application que si elle satisfait un ensemble de conditions ou à l'inverse déclarer que la permission est accordée à toute application sauf celles qui satisfont cet ensemble de conditions.

À l'exécution, Saint permet aussi de contrôler les communications entre applications en définissant grâce aux mêmes types de politique de sécurité. L'émetteur et le destinataire peuvent tout deux définir cette politique. Comme à l'installation, l'application définit un comportement par défaut (autoriser ou bloquer) et les conditions qui y font exception. La différence par rapport à l'installation est que l'application peut définir des contextes d'exécution comme condition dans la politique. Ces contextes peuvent par exemple être des plages horaires, l'état de connexion au réseau ; la localisation du téléphone, etc.

Saint offre aux applications un contrôle plus fin sur l'attribution et l'usage des permissions qu'ils déclarent pour protéger leurs composants. À l'exécution, Saint souffre cependant des mêmes limitations que Porscha [79]. En effet, la protection offerte par Saint se limite à l'accès direct au composant de l'application protégée par la politique de sécurité. Si une application autorisée par la politique de sécurité est par exemple utilisée lors d'une attaque par délégation pour accéder au composant protégé, Saint ne verrait pas que l'accès est en fait réalisé par une autre application qui elle n'est peut-être pas autorisée par la politique de sécurité.

2.5.2.3 Quire

Pour protéger les accès à des composants sensibles, les travaux précédents se contentent de contrôler l'accès direct au composant. Ces solutions sont ainsi inefficaces pour lutter contre les attaques par délégation. Dans [42], Dietz et al. proposent Quire une extension de sécurité à Android dont le but est de détecter et bloquer les attaques par délégation. Quire modifie les mécanismes de communication entre applications afin qu'à chaque requête émise, l'émetteur puisse joindre la chaine d'appel ayant mené à l'émission de cette requête et que le destinataire de la requête puisse vérifier que tous les éléments de la chaine d'appel ont les permissions nécessaires pour demander le traitement de la requête. Par exemple, si une application A envoie une requête à une application B qui à la suite de la réception de la requête envoie une requête à l'application C alors la chaine d'appel une fois arrivée à C est A → B. Si C à la réception de la requête effectue une opération sensible, elle peut vérifier si les applications dans la chaine d'appel ayant mené à l'exécution cette opération ont les permissions nécessaires. Aini, en supposant que B ait la permission nécessaire pour envoyer la requête et que A ne l'ait pas, C peut décider en connaissance de cause si oui ou non

elle exécute l'opération demandée. Si l'opération sensible exécutée par C est par exemple l'édition de la liste de contact, alors C vérifiera que les applications A et B possèdent la permission pour éditer les contacts (`WRITE_CONTACTS`).

2.5.2.4 IPC Inspection

Porter et al. ont adopté une approche similaire à Quire pour bloquer les attaques par délégation. Dans [49], ils présentent IPC Inspection un mécanisme qui contrôle les communications entre applications sous Android. Quand une application reçoit un message, le système considère que ses permissions sont réduites à l'intersection de l'ensemble de ses permissions avant la réception du message et l'ensemble des permissions de l'émetteur. En agissant ainsi, le destinataire ne peut plus effectuer d'action sensible à la demande de l'émetteur si ce dernier n'a pas la permission nécessaire pour effectuer l'action. Le destinataire des messages voit ainsi ses permissions à l'exécution se réduire au fur et à mesure qu'elle reçoit des messages de différente applications avec des permissions différentes.

Quelques règles existent afin d'éviter que les applications destinataires des messages ne perdent leur permission définitivement et deviennent inutilisables. Les applications systèmes font partie de la base de confiance. Tout message provenant de ces applications n'entraînent donc aucune réduction des permissions. Un `intent` peut servir à demander d'exécuter une action particulière et à être notifié à la fin d'exécution de cette tâche. La notification se fait par l'envoi d'un `intent` à l'émetteur du précédent message. Lors de la réception de la notification, le système considère qu'il n'y a pas risque d'attaque et n'effectue aucune réduction de permission. Enfin, IPC Inspection force l'usage de multiples instances d'une même application pour traiter les messages qui lui sont adressés. Le destinataire d'un message a ainsi une instance avec ses permissions d'origine et d'autres instances qui traiteront chacun des messages et dont les permissions seront réduites.

2.5.2.5 Prévention des attaques par délégation et par collusion

Quire [42] et IPC Inspection [49] ne ciblent que les attaques par délégation. De plus, le seul mécanisme de communication pris en compte est celui basé sur `binder` en utilisant les `intents`. Dans [30], Bugiel et al. présentent une nouvelle extension de sécurité à Android qui est paramétrée par une politique de sécurité et dont le but est de bloquer les attaques par délégation ainsi que les attaques par collusion. Contrairement aux travaux précédents, ils prennent en compte la possibilité que la communication entre les applications se fasse également à travers les mécanismes standard de communication sous Linux (fichiers et sockets). Ils étendent un framework de sécurité pour Android développé dans un travail antérieur [29] afin d'observer les interactions entre les applications, composants du système (ContentProvider et services), fichiers et sockets. À chaque fois qu'une interaction est observée, ils vérifient qu'il n'y a pas un risque d'attaque. La détection des attaques utilisent une représentation sous forme

de graphe des interactions entre les applications, les fichiers, les sockets et des composants du système (ContentProvider et services). La politique de sécurité décrit des propriétés sur ce graphe qui permettent de statuer si une interaction correspond à une attaque ou non. Par exemple, une attaque par délégation est décrite par le fait qu'il existe un chemin à partir du nœud représentant une application A vers le nœud représentant une application B tel que B possède des permissions critiques que A ne possède pas. Une attaque par collusion est décrite par le fait qu'il existe une chemin entre les nœuds de deux applications A et B tel que l'union des permissions de A et B soit critique. Les auteurs ne définissent pas ce qu'est un ensemble critique de permissions et laisse plutôt le soin de le définir à l'utilisateur qui définit la politique de sécurité.

Contrairement à Quire, Bugiel et al. laissent le choix à l'utilisateur de définir ce qu'est une attaque via la politique de sécurité, ce qui apporte à la fois un avantage et un inconvénient à l'approche. La possibilité de définir avec précision les scénarios d'attaque a l'avantage de limiter les faux positifs lors de l'exécution du système, c'est-à-dire des communications jugées comme dangereuses alors qu'elles ne le sont pas. Un des cas souvent ignoré dans les travaux précédents est qu'une application puisse intentionnellement fournir un service à une autre application. L'inconvénient dans cette approche est que l'utilisateur n'a pas forcément les compétences nécessaires pour définir une bonne politique de sécurité ce qui pourrait être exploiter par une application malveillante afin d'échapper à la détection.

2.5.3 Abus des permissions

2.5.3.1 Kirin

L'approche basique utilisée par les applications malveillantes sous Android est de demander toutes les permissions qui leur sont nécessaires pour effectuer leur tâche malveillante. Par exemple, les applications espions dont le but est de tracer l'utilisateur du téléphone demande l'accès aux données de géolocalisation et l'accès à internet.

Afin de bloquer ce type d'attaque, Enck et al. proposent un système nommé Kirin dans [47]. Le but de Kirin est de vérifier qu'une application n'a pas un ensemble de permissions jugé dangereux. Par exemple, l'accès aux données de géolocalisation et à internet permet de faire fuir les déplacements de l'utilisateur de téléphone. Ainsi, Kirin vérifie à l'installation les permissions demandées par une application. Si elle contient un ensemble dangereux, l'installation est bloquée. Il appartient à l'utilisateur ou à l'administrateur de l'appareil de définir les ensembles de permission dangereux.

2.5.3.2 Woodpecker

Les permissions filtrent l'accès aux ressources sensibles sous Android. Une application sans la permission `READ_SMS` ne peut par exemple lire la base de données des `SMS`. Dans [54], Grace et al. ont analysé les applications livrées avec

8 téléphones Android et montré que certaines exposaient sans aucune restriction les ressources sensibles. Les auteurs ont identifié deux méthodes.

La première méthode, qu'ils disent explicite, consiste à exposer les méthodes pour accéder aux ressources sensibles via des interfaces publiques dont l'accès est plus laxiste que celui de la méthode protégée. Ils considèrent comment interface les composants exposés publiquement à d'autres applications. La deuxième méthode, cette fois-ci implicite, consiste à partager le même identifiant d'utilisateur entre deux applications. Cela est possible en associant la même valeur à l'attribut `sharedUserId` dans le fichier `AndroidManifest.xml` des deux applications. Les deux applications tournant avec le même identifiant utilisateur, elles se retrouvent ainsi à l'exécution avec l'union des permissions que les deux applications possèdent.

Grace et al. ont ainsi développé Woodpecker un outil pour analyser les applications et trouver d'éventuelles expositions de ressources sensibles. Woodpecker définit les points d'entrée de l'application à partir de son fichier `AndroidManifest.xml`, construit son graphe de flux de contrôle (*Control Flow Graph* en anglais) et à partir du CFG calcule tous les chemins d'exécution possibles. L'outil considère qu'une ressource sensible est exposée explicitement si un chemin d'exécution partant d'une interface non protégée contient un appel à une fonction sensible et qu'aucun contrôle n'est fait avant l'appel. Pour le deuxième cas, une alerte est levée si une application indique partager son identifiant utilisateur et s'il existe un appel à une fonction sensible dans le code de l'application telle que la partie contenant cette fonction puisse être atteinte.

2.5.3.3 Aurasium

La plupart des approches visant à contrôler le comportement des applications se font en modifiant le système. Dans [105], Xu et al. proposent cette fois-ci de laisser le système intact et d'ajouter le code nécessaire au contrôle dans l'application. Ils ont ainsi développé un service appelé Aurasium [8] qui prend en entrée une application et retourne sa version renforcée. Cette version renforcée intègre du code interceptant les appels aux fonctions sensibles sous Android afin de renforcer une politique de sécurité. Les fonctions concernées servent à accéder à des ressources sensibles sur le téléphone et internet, et à exécuter des opérations potentiellement dangereuses telles que le chargement d'une bibliothèque ou l'exécution d'un binaire.

Bien que ce genre de fonctions soient définies au sein de l'API Java d'Android, Aurasium intercepte leur appel à un niveau plus bas, plus précisément au niveau des fonctions de la bibliothèque C du système et de la machine virtuelle dalvik. Il existe par exemple différentes fonctions dans l'API pour communiquer sur le réseau. Cependant, au niveau système tout cela se traduit par un ensemble restreint d'appels système tels que `connect` et `sendmsg`. Il en est de même pour les IPCs basés sur le `binder` qui se font via l'appel système `ioctl /dev/binder/`. Lorsque l'appel à une fonction sensible est détectée, l'application

8. `http://aurasium.com`

en informe l'utilisateur et demande à l'utilisateur de valider l'appel s'il souhaite qu'il se poursuive.

Pour évaluer la capacité de leur service à instrumenter les applications et à fournir une version toujours fonctionnelle, ils ont soumis un ensemble de 3491 applications de Google Play et 1260 échantillons de malware à leur service. Nous entendons par version fonctionnelle, une application qui se lance sans erreur au démarrage et qui à l'exécution montre bien des interceptions à des appels de fonction sensible. Leur évaluation a montré qu'Aurasium était capable de fournir une version fonctionnelle des applications dans plus de 99% des cas. Aucune évaluation ne permet cependant d'apprécier la capacité d'Aurasium à intercepter les appels de fonction correspondant aux attaques sur un système.

Nous avons présenté dans cette section différents travaux inhérents à la sécurité d'Android. Partant du constat des limites des mécanismes de sécurité Android, principalement celles liées à l'accès aux ressources sensibles et aux communications entre applications, les auteurs de ces travaux ont proposé des extensions de sécurité à Android et des outils d'analyse d'application. Ces extensions et outils ont pour but de détecter les tentatives exploitant les limites du mécanisme de sécurité du système ainsi que les risques de sécurité introduit par les mécanismes de communication Android. Parmi les attaques ciblées, ils ciblent principalement le vol de données sensibles, l'usage malveillant des permissions accordées aux applications, les attaques par délégation et les attaques par collusion. Ces travaux concernent ainsi des classes d'attaque connue et essayent de les détecter ou les prévenir.

Dans cette thèse, nous avons une approche différente. Au lieu de détecter les malware en nous basant sur les limites connues du mécanisme de sécurité d'Android, nous souhaitons dans un premier temps caractériser les malware (comment leur attaque fonctionne) et nous baser sur ce que nous aurons appris pour les détecter. Notre approche est basée sur les flux d'information que causent les malware dans le système. Dans la section qui suit, nous introduisons ainsi la notion de flux d'information et présentons quelques travaux utilisant le suivi de flux d'information pour détecter des attaques.

2.6 Suivi de flux d'information

Suivre les flux d'information consiste à observer comment les informations se propagent dans un environnement donné. En sécurité, le suivi de flux d'information est utilisé pour détecter des attaques visant soit la confidentialité soit l'intégrité de données dans un environnement donné. En plus de détecter des attaques, il peut également servir à les bloquer. Il s'agit dans ce cas de contrôle de flux d'information.

2.6.1 Suivi de flux au sein d'une application

Suivre les flux d'information au sein d'une application permet d'avoir une vue fine de la manière dont les informations sensibles sont traitées par un programme. À l'exception du langage Perl, les langages de programmation n'intègrent cependant pas de mécanisme de suivi ou de contrôle de flux d'information. Divers travaux se sont ainsi concentrés sur le suivi et le contrôle de flux dans les applications soit en proposant une extension au langage de programmation utilisé soit en instrumentant l'environnement dans lequel l'application tourne.

Les applications Android sont principalement écrites en Java. Dans [74, 73], Myers et Liskov présentent un modèle de suivi de flux d'information qu'ils implémentent dans Jif [75], une extension du langage Java capable de contrôler les flux d'information à l'intérieur d'une application. Techniquement, Jif est un préprocesseur effectuant une analyse statique du code source pour contrôler la conformité des flux d'information dans un programme par rapport à une politique de flux. La politique est représentée par des labels qui sont associés aux variables et fonctions dans le code. À chaque objet est assigné un label où sont déclarés l'ensemble des propriétaires de l'information et les entités que chaque propriétaire autorise à accéder à l'information. Étant donné que chaque propriétaire déclare une liste de lecteurs autorisés, la liste des lecteurs effectifs est ensuite calculée en faisant l'intersection de la liste que chaque propriétaire a déclarée. Le label $L = \{o1 : r1, r2; o2 : r2, r3\}$ signifie par exemple que l'objet auquel il est attaché contient des informations dont $o1$ et $o2$ sont propriétaires. $o1$ et $o2$ n'autorisent respectivement que $r1$ ou $r2$ et $r2$ ou $r3$ à accéder à l'information. Le lecteur effectif est donc ici $r2$. Lors prétraitement du code, Jif, qui est un préprocesseur Java, calcul les flux d'information dans le programme et vérifie que les politiques de sécurité associées aux variables sont vérifiées.

L'approche proposée par Myers et Liskov est une approche statique qui consiste à annoter le code source des applications puis à l'analyser pour contrôler les flux d'information dans l'application. Un des avantages de l'analyse statique par rapport à l'analyse dynamique est qu'il couvre tous les chemins d'exécution possibles et permet en une seule analyse d'identifier toutes les violations à une politique de flux d'information. Une limite à cette approche est cependant la nécessité d'avoir accès au code source de l'application pour l'analyser, ce qui est rarement le cas pour les applications Android et limite l'usage de cette approche. Le code des applications Android est livré sous la forme de `dalvik bytecode` et il est parfois obfusqué afin d'empêcher des entités tierces de retrouver le code original à partir duquel le `dalvik bytecode` a été généré. Ce `bytecode` est obtenu en compilant dans un premier temps le code source de l'application en `bytecode Java` puis en compilant ce dernier en `bytecode` compréhensible par la machine virtuelle Dalvik.

Si les travaux précédents s'attachent à ajouter une extension de contrôle de flux d'information dans le langage de programmation et contrôler les flux d'information de manière statique, d'autres se sont penchés sur une modification de l'environnement d'exécution pour suivre et contrôler les flux d'information à

l'exécution des applications. Dans [46], Enck et al. proposent TaintDroid, une version modifiée d'Android avec un mécanisme de suivi de flux d'information à l'exécution des applications. Leur but était d'étudier si les applications Android font fuir des données sensibles du téléphone via le réseau ou par SMS. Ils ont défini dans leur outil un ensemble d'information à surveiller (ex : identifiants de l'appareil et données de géolocalisation) et sélectionné les 30 applications Android les plus populaires pour les analyser. Ils ont montré que plus de 2/3 d'entre elles faisaient fuir des données sensibles, principalement les identifiants du téléphone. Pour suivre les flux d'information, Enck et al. ont modifié la machine virtuelle Dalvik. Ils ont implémenté une méthode dite de tainting qui consiste à marquer les objets selon la nature de leur contenu. Dans la machine virtuelle Dalvik, les objets pouvant contenir des informations sont les variables d'une méthode, ses paramètres et les champs d'une classe ainsi que des ses instances. Lorsqu'un flux est obseré, les marques associées aux objets dont le contenu a été modifié est mis à jour pour prendre en compte la nature de leur nouveau contenu.

Contrairement à Myers et Liskov, Enck et al. ont opté pour une approche dynamique pour suivre les flux d'information au sein d'une application durant son exécution. À cause de la nature de l'analyse (dynamique), une des principales limitations de TaintDroid est l'impossibilité d'observer tous les flux d'information possibles en une seule analyse. Seuls les flux d'information explicites qui ont lieu durant l'exécution de l'application sont détectés. Si TaintDroid a une vue plus limitée des flux d'information dans une application, il possède cependant un avantage non négligeable par rapport aux approches statiques qui est qu'il ne nécessite pas l'accès au code source des applications analysées. Il est donc plus adapté pour analyser les applications destinées à tournées dans un environnement tels qu'Android.

2.6.2 Suivi de flux au niveau système

Un autre niveau d'observation des flux d'information est le système d'exploitation où les objets du système tels que les processus et les fichiers sont vus comme des conteneurs d'information et les interactions entre eux des flux d'information. Dans [109, 110], Zeldovich et al. proposent HiStar, un système d'exploitation qui intègre un mécanisme de contrôle de flux d'information au niveau système. Le but de HiStar est de fournir un système capable de bloquer les attaques visant à voler ou corrompre les données sur le système. Pour suivre et contrôler les flux d'information, HiStar utilise les labels Asbetos [45]. Un label établit une correspondance entre une catégorie d'information et un niveau marquage. Par exemple, le label $L = \{w3, c2, 1\}$ signifie que le niveau associé aux catégories w et c sont respectivement 3 et 2. Le niveau par défaut est 1. Un label est associé à chaque objet du système pouvant contenir de l'information. Le label représente pour chaque objet son niveau d'accès pour les différentes catégories d'information. Lorsqu'un flux d'information d'un objet A vers un objet B a lieu, HiStar vérifie que le niveau d'accès de B pour les catégories d'information définies dans le label de A est supérieur ou égal au niveau d'accès de

A pour ces mêmes catégories. Si c'est le cas, le flux est autorisé. Dans le cas contraire, il est bloqué. Par exemple, si le label de A vaut $\{w3, c2, 1\}$ et que celui de B vaut $\{w2, c3, h2, 1\}$ alors B ne peut accéder au contenu de A car le niveau d'accès de B aux à la catégorie d'information w, ici 2, est inférieur au niveau d'accès de A, ici 3, à la même catégorie.

Pour contrôler les flux d'information dans le système, HiStar associe des niveaux d'accès aux différents objets du système. Contrairement à TaintDroid ou AppFence, il se contente d'établir des niveaux d'accès aux informations mais ne suit pas leur propagation dans le système. Cela limite son utilité pour analyser comment les données d'une application se propagent dans le système, ce qui est l'un des nos objectifs.

Dans cette thèse, nous utilisons un moniteur de flux d'information au niveau du système pour observer les flux d'information. Ce moniteur est la version Android d'un moniteur de flux d'information pour Linux, Blare [37], dont la version actuelle a été développée par Christophe Hauser. Plus précisément, Blare est un outil de détection d'intrusion pour les systèmes Linux paramétré par une politique de flux d'information. La politique définit les flux autorisés dans le système et toute violation de cette politique constitue une intrusion. Pour suivre et contrôler les flux d'information, Blare utilise deux labels, *itag* et *ptag*, qui sont attachés aux objets du système. L'*itag* représente la contamination courante d'un objet et le *ptag* représente sa contamination maximale autorisée. L'*itag* d'un objet est ainsi mis à jour à chaque fois qu'un flux d'information modifiant le contenu de l'objet a lieu tandis que le *ptag* est défini lors de la création de la politique de flux. Lorsqu'un flux d'information d'un objet A vers un objet B a lieu, Blare considère que les informations de A vont dans B et vérifie si l'*itag* de A est autorisé par le *ptag* de B. Si c'est le cas, il met à jour l'*itag* de B pour prendre en compte son nouveau contenu. Sinon il lève une alerte. Blare ne prévient pas les intrusions mais les détecte. Il ne bloque pas le flux en cas de violation d'une politique mais se contente de lever une alerte. Comme HisTar, le niveau d'observation de Blare se limite au niveau système. Il voit les objets du système comme des boîtes noires et effectue une surapproximation des flux observés. Quand un flux d'un objet A vers un objet B est observé par Blare, il considère que toutes les informations dans A se propagent dans B. Cette vision est moins fine que ce que proposerait un suivi de flux au niveau des applications. Cependant il a l'avantage de ne pas dépendre d'un langage en particulier tel que le Java et ne se limite pas aux instances d'une machine virtuelle. Il a ainsi une vue complète des flux d'information directs entre les objets du système par rapport à TaintDroid qui est limité aux flux impliquant uniquement les applications écrites en Java.

2.6.3 Suivi de flux au niveau hardware

Dans [108], Yin et al. proposent un environnement virtuel basé sur Quemu d'analyse dynamique d'application Windows appelé Panorama. Panorama suit les flux d'information au niveau machine tout en ayant connaissance des informations liées aux objets du système tels que les processus et les fichiers. Le but du

travail réalisé est de déterminer si une application accède à une donnée sensible du système et si c'est le cas comment elle l'utilise. Les données sensibles peuvent être des données frappées au clavier, des données du disque ou le contenu de paquets réseaux. Dans le cadre de l'analyse, ces données sont supposées ne pas être accédées par l'application analysée. Pour analyser une application, ils installent ainsi l'application dans l'environnement d'analyse, introduisent des données sensibles dans le système, et suivent comment ces données se propagent. À partir des flux observés, ils construisent un graphe de flux d'information représentant comment les données sensibles se sont propagées dans le système. Les nœuds et les arcs du graphe représentent respectivement les objets du système et les flux d'information entre eux. L'application analysée est considérée comme malveillante si un nœud la représentant est présent dans le graphe. Cela signifie qu'à un moment donné, l'application a accédée aux données sensibles. Si c'est le cas, ils analysent ensuite où les données se propagent à partir du nœud représentant l'application analysée.

Dans des travaux plus récents, Yin et al. ont proposé DroidScope [106] qui est un environnement d'analyse d'application Android qui est l'équivalent de Panorama pour Android. DroidScope suit également les flux d'information au niveau hardware mais contrairement à Panorama, il n'a pas seulement accès aux informations liées aux objets du système mais également à des données liées à l'exécution des applications dans la machine virtuelle Dalvik. Il a ainsi une vue plus fine de ce qui se passe dans une application telle que les méthodes qui sont utilisées pour accéder à une information.

Suivre les flux d'information au niveau hardware permet d'avoir une vue fine des flux d'information s'opérant sur un système. Le niveau de granularité pour le stockage et l'accès à une information peut-être réduit à un octet ou un bit, ce qui est l'unité de stockage des données. Avoir une telle granularité a cependant un coût élevé en terme de temps d'exécution qui limite cette approche aux environnements virtuels d'analyse d'application. Les auteurs de Panorama ont mesuré un temps d'exécution des applications de 30 à 40 fois plus lents que leur temps d'exécution normal. Notre but est de capturer le comportement malveillant des malware Android et de les détecter. Sachant que certains malware essayent de détecter la présence d'environnement virtuels pour échapper à toute détection, ce niveau d'observation ne nous paraît donc pas comme un choix viable.

Il existe différents niveaux d'observation pour suivre les flux d'information sur un système : application, système d'exploitation et hardware. Dans ce thèse, nous optons pour un suivi de flux au niveau du système d'exploitation afin d'avoir une vue complète des flux d'information entre les différents composants du système. Suivre les flux d'information à l'intérieur des applications a une meilleure granularité mais suppose que nous ayons, soit accès au code source de toutes les applications à analyser, soit que toutes les applications soient écrites en Java et interprétées par la machine virtuelle Dalvik. Aucune de ces suppositions ne s'avère être vraie dans la réalité. Suivre les flux d'information au niveau

du hardware apporte également une vue plus fine et contrairement au niveau application ne nécessite pas d'accès au code source ou l'usage d'un seul langage pour écrire toutes les applications. Cependant, comme nous l'avons écrit précédemment, il n'est adapté qu'aux environnements virtuels, ce qui pose le problème de la détection de l'environnement d'analyse [68]. Dans cette thèse, nous avons ainsi fait le choix de suivre les flux d'information au niveau système afin d'être indépendant de la disponibilité du code source des applications et du langage utilisé pour les développer tout en ayant une vue complète des flux d'information entre les objets du système. À partir des flux d'information observés dans le système, nous proposons de calculer le profil d'un malware et apprendre son comportement. L'approche est similaire à de l'apprentissage. Dans la section qui suit, nous présentons donc comment des méthodes d'apprentissage ont été appliqués pour classifier et détecter des malware.

2.7 Classification et détection de malware

La détection de malware se fait généralement en utilisant des signatures ou des profils comportementaux. La signature d'un malware est un ensemble de propriétés communes aux fichiers des échantillons du malware. Dans le cas des applications Android, ces fichiers sont les `apk`. Le profil comportemental d'un malware est un ensemble de propriétés communes à l'exécution des échantillons du malware. Ces signatures et profils sont construits à partir des informations obtenues en analysant, statiquement ou dynamiquement, les échantillons de malware. La tendance actuelle consiste à utiliser des méthodes d'apprentissage afin de classifier et détecter des malwares. Dans notre cas, nous souhaitons extraire le profil d'un malware et son comportement à partir des flux d'information qu'ils causent dans le système. Dans ce qui suit, nous présentons ainsi quelques travaux ayant adopté cette approche. L'apprentissage consiste à apprendre des propriétés liées à un jeu de données.

Il existe différents travaux sur l'analyse statique d'application afin de détecter des instances de malware (ex : [35] et [58]). Les signatures sont des propriétés liées au malware telles que des chaines de caractères, des ressources dans l'application, des appels à des fonctions spécifiques, ou simplement l'empreinte d'un fichier. Sous Android, Arp et al. ont proposé DREBIN [22] pour détecter les échantillons de malware Android sur les téléphones. Les propriétés prises en compte lors de l'analyse d'une application sont les permissions demandées, les composants matériels requis, ses composants, les filtres d'`intent` pour activer les composants, les appels aux fonctions sensibles, les permissions utilisées, les appels de fonction suspects et les adresses de serveur présents dans le code de l'application. Pour classifier les applications, ils utilisent une machine à vecteurs de support [41, 48] qui est une méthode d'apprentissage supervisée. Un apprentissage supervisé consiste à calculer à partir d'ensembles distincts un modèle permettant de définir l'appartenance d'un élément à un de ces ensembles. À partir de deux ensembles distincts d'applications, l'une contenant des applications malveillantes et l'autre contenant des applications bénignes, ils calculent

ainsi un modèle de détection qui différencie les applications malveillantes des applications bénignes. Ce modèle est ensuite utilisé sur le téléphone afin de détecter les applications malveillantes. L'évaluation effectuée par les auteurs de DREBIN sur la capacité de détection montre que leur outil atteint un taux de détection de 93.90% sur un ensemble de 1834 échantillons de malware. Le taux de détection définir pourcentage d'échantillons détectés sur tout le jeu de données. Le résultat est proche des meilleurs résultats obtenus par les produits anti-virus qu'ils ont testé sur le même jeu de données. Les meilleurs taux de détection sont de 96.41% et de 93.71% tandis que le pire est de 3.99%.

La détection par analyse statique montre cependant ses limites lorsque les développeurs de malware utilisent des techniques d'obfuscation [21, 23, 70] afin de cacher la véritable nature de leurs application ou des techniques de polymorphisme et de métamorphisme [50] afin de changer l'aspect du code des différentes instances d'un malware. Les développeurs du malware Obad.a [98] utilisent par exemple des chaines aléatoires pour les noms de variables, méthode et classes. Ils utilisent également des fonctions de chiffrement afin de masquer les chaines de caractères utilisées dans le malware et ces fonctions changent d'un échantillon à l'autre.

Contrairement à l'approche statique, l'approche dynamique repose uniquement sur l'exécution des applications pour détecter les échantillons de malware et n'est pas ainsi affectée par ces méthodes d'évasion. Dans [67], Bayer et al. appliquent une méthode d'apprentissage non supervisée afin de classifier les applications et détecter les échantillons de malware. L'apprentissage non supervisée considère qu'il n'y a pas de groupement prédéfinis des éléments dans le jeu de données et calcule lui-même les regroupements possibles de ces éléments.

Les applications à classifier sont analysées dans Anubis [69] afin d'en extraire un profil. Le profil est constitué des diverses interactions avec le système tels que les accès aux fichiers et registres ainsi que les communications réseaux. Ils appliquent également une méthode de *tainting* afin d'observer comment les informations issues du système sont utilisées dans les prochains appels système. Cela s'avère utile pour filtrer les informations à extraire. Par exemple, si le programme utilise la date du système pour générer le nom d'un nouveau fichier, il ne sert à rien de stocker le nom car il sera différent à chaque exécution.

Dans [90], Rieck et al. proposent d'utiliser les interactions entre les applications analysées et le reste du système afin de les classifier et de se servir du modèle obtenu lors de l'apprentissage pour détecter d'autres échantillons de malware. Pour obtenir les interactions des applications avec le reste du système, ils exécutent les applications dans CWSandbox [101], un environnement d'analyse dynamique d'application. Ils construisent ensuite un vecteur représentant le comportement de l'applications analysée à partir des interactions observées où chaque dimension représente une suite d'interaction et utilisent ces vecteurs afin de classifier de manière non supervisée les applications analysées. Les classes obtenues à la suite de cette étape servent ensuite de modèle afin de détecter de nouvelles instances de malware.

Rieck et al. ont comparé la qualité des classes crées durant l'apprentissage rapport à celle de Bayer et al. et ont montré que leur approche obtient un

meilleur score, 0.950, par rapport à celle de Bayer et al., 0.881. Le score calculé, appelé F-Score [8], combine à la fois la qualité de la précision ainsi que du rappel des classes calculées durant l'apprentissage. La précision mesure la correspondance entre les classes calculées et les familles de malware tandis que le rappel mesure l'éparpillement des échantillons d'un malware dans plusieurs classes. Une précision élevée signifie que chaque classe correspond à une famille de malware. Un rappel élevé signifie que chaque famille de malware correspond à une classe. La valeur maximale du F-score est de 1 tandis que la pire est de 0. Plus le score est élevé, plus la qualité de l'apprentissage est meilleure.

Les travaux présentés ont monté que l'apprentissage pouvait servir à calculer des modèles afin de classifier et détecter des malware. Cet apprentissage est effectué à partir des propriétés liées aux fichiers contenant les applications à analyser ou de leurs traces d'exécution. Dans notre cas, l'apprentissage est effectué sur les flux engendrés par les échantillons de malware dans le système. Nous exécutons ainsi chaque échantillon de malware pour capturer ces flux d'information. Un des buts de notre approche est d'identifier le comportement malveillant des malware. Pour cela, il faut capturer ce comportement durant l'analyser. Pour nous en assurer, nous avons analysé statiquement les échantillons de certains malware pour identifier les évènements déclenchant l'exécution du code malveillant. Dans la section qui suit, nous présentons donc un ensemble d'outils liés à l'analyse d'applications Android.

2.8 Analyse d'applications Android

Analyser une application a pour but d'extraire des informations liées à l'application telles que son comportement, les ressources qu'elle utilise et les évènements attendus pour exécuter une partie de son code.

2.8.1 Désassembleur, décompilateur

Afin de comprendre le fonctionnement d'une applications sans avoir à l'exécuter, un analyste a souvent recours à un désassembleur ou un décompilateur. Désassembler le code d'une application consiste à le retranscrire dans un langage bas niveau dont les instructions et celles de l'architecture censée interprétée le code à désassembler ont une forte correspondance. L'architecture interprétant le code d'une application Android est la machine virtuelle Dalvik. Parmi les outils pour désassembler les applications Android, nous pouvons citer dedexer [6], apktool, qui utilise smali [13], Androguard [97] et IDA Pro [9]. Le listing 2.4 est un extrait du code de l'application JetBoy disponible dans le SDK Android. Le listing 2.5 est la partie correspondante, écrite en smali, après avoir désassemblé l'application avec apktool. La syntaxe des langages utilisés par ces outils est assez explicite pour comprendre ce que fait l'application et avoir une idée de l'aspect du code Java d'origine. Nous remarquons par exemple que l'instruction à la ligne 4 dans le listing 2.4 correspond aux instructions de la ligne 24 à 28 du listing 2.5.

```
1  public void onClick(View v) {
2      // this is the first screen
3      if (mJetBoyThread.getGameState() == JetBoyThread.STATE_START) {
4          mButton.setText("PLAY!");
5          mTextView.setVisibility(View.VISIBLE);
6
7          mTextView.setText(R.string.helpText);
8          mJetBoyThread.setGameState(JetBoyThread.STATE_PLAY);
9      }
10     ...
11 }
```

Listing 2.4 – Extrait du code de l'application JetBoy fourni avec le SDK Android

Décompiler une application revient à le retranscrire dans un langage haut niveau (ex : Java et C). Les applications Android étant écrites en Java, le but de la décompilation sera ici d'obtenir un code Java à partir du *dalvik bytecode*. Parmi les outils notables nous pouvons citer Androguard et la combinaison dex2jar [43] et JD-Gui [10]. Androguard intègre un décompilateur natif, `dad`, mais il peut également faire appel à d'autres décompilateurs. dex2jar est un outil qui transforme le *dalvik bytecode* en *java bytecode* et JD-Gui est un outil graphique de décompilation de *java bytecode*. Décompiler une application Android en optant pour la combinaison de ces outils revient ainsi à extraire le fichier `classes.dex` de l'`apk`, créer avec dex2jar un fichier `jar` (Java Archive) et à décompiler ce dernier avec JD-Gui. Un avantage que JD-Gui a sur Androguard est la possibilité d'exporter le résultat de la décompilation dans les fichiers `.class`. Le listing 2.6 liste le code produit par la décompilation de l'application JetBoy avec Androguard. Une différence notable avec le code orignal est l'usage des valeurs à la place des variables constantes présentes dans le code original.

Si le résultat de la décompilation a le mérite d'être dans un langage haut niveau, cette opération intègre cependant le risque d'une erreur de transcription. Lors de la décompilation de certaines applications, nous avons ainsi détecter des erreurs dans le résultat de la décompilation pour les deux outils.

2.8.2 Comparaison d'applications

Une autre fonctionnalité intéressante d'Androguard est sa capacité à évaluer la similarité entre deux applications. Androguard compare le code des deux applications et calcule quelles sont les méthodes qui sont identiques, similaires et celles qui sont présentes dans l'une mais pas dans l'autre. L'un des cas d'usage de cette fonctionnalité est par exemple de trouver le code rajouté à une application entre deux versions. Elle peut également servir à identifier les applications volées à leur auteur original.

2.8.2.1 Extraction de métadonnées

L'analyse peut également avoir pour but d'extraire des métadonnées liées à l'application. Dans le cas de applications Android, ces données sont généralement le contenu du fichier `AndroidManifest.xml`. Ce fichier contient différentes informations : permissions utilisées par l'application, son numéro de version, la liste de ses composants, etc. L'accès à son contenu ne nécessite pas forcément un outil particulier car il s'agit d'un fichier XML. Androguard ou les outils d'analyse dynamique d'application tels que Andrubis propose cependant d'extraire automatiquement ces donnés.

2.8.2.2 Scanner de virus

Il existe plusieurs scanners de virus sous Android dont certains sont disponibles directement sur Google Play. Il existe cependant pour les analystes des services permettant d'analyser les applications Android pour détecter les échantillons de malware sans avoir à les analyser avec une application dans un téléphone. VirusTotal [14] est un exemple de ce type d'outil en ligne. Pour effectuer une analyse, les utilisateurs soumettent le fichier à analyser via une page de soumission. Chaque fichier soumis est ensuite analysé par différents produits anti-virus afin de détecter si oui ou non il est malveillant. VirusTotal utilise à ce jour 49 produits anti-virus. Le résultat des analyses de chaque anti-virus est ensuite retournée à l'utilisateur. AndroTotal [2] est un autre outil similaire à VirusTotal. Comme son nom l'indique, AndroTotal est uniquement dédié à l'analyse d'application Android alors que VirusTotal n'impose aucune restriction sur le type de fichier à analyser. Ce type d'outil peut s'avérer utile pour identifier rapidement des échantillons de malware. Zhou et al. ont par exemple utilisé VirusTotal afin de classifier les échantillons de malware qu'ils ont analysé dans [113].

2.8.2.3 Analyse dynamique d'application

Analyser dynamiquement une application consiste à l'exécuter pour observer son comportement. L'exécution se fait généralement à l'intérieur d'un environnement virtuel simulant un environnement d'exécution tel qu'un système d'exploitation ou une machine. Ils existent plusieurs environnements d'analyse d'application Android. Certains d'entre eux sont disponibles via leur code source (ex : DroidBox) ou en tant que service en ligne (ex : Andrubis). Ils ont souvent une base commune d'informations surveillées : permissions utilisées, appels de fonction sensible de l'API Android, composants de l'application qui ont été exécutées, accès aux systèmes de fichier, communications réseaux, et émissions d'appel et SMS.

Pour observer toutes ces informations, les auteurs de ces outils ont recours à différentes techniques : réécriture de code, suivi de flux d'information, observation des appels système, introspection de l'environnement virtuel d'exécution et stimulation d'entrée.

La réécriture de code consiste à injecter le code nécessaire à l'analyse dans l'application avant de l'exécuter. Durant l'exécution, le code injecté renseignera sur les comportements d'intérêt. Les auteurs de DroidBox ont par exemple opté pour cette méthode afin d'observer les appels aux différentes fonctions sensibles de l'API Android.

Le suivi de flux d'information permet d'observer comment une information se propage dans un environnement donné. Un appareil Android contient un certain nombre d'informations sensibles (ex : liste de contact) dont l'usage par une application aurait une certaine utilité pour un analyste. Andrubis, AppsPlayground et DroidBox intègrent ainsi TaintDroid afin de suivre les flux d'information dans les applications Android pour détecter toute fuite d'information. DroidScope a son propre moniteur de flux d'information

Ces environnements d'analyse dynamique d'application peuvent être fournis sous deux formes. Soit leurs auteurs fournissent un accès en tant que service, l'analyse est effectuée en soumettant l'application à analyser et l'utilisateur récupère le résultat de l'analyse sous forme de page web ou de fichiers XML ou JSON, soit ils fournissent l'environnement en téléchargement afin que l'utilisateur l'exécute sur sa proche machine.

DroidBox [7] est un environnement d'analyse dynamique d'application Android à télécharger et exécuter sur sa propre machine. Durant l'exécution de l'application, l'outil enregistre les communications réseaux effectuées par l'application, les accès aux fichiers, les services lancés, les composants Broadcast Receiver, les classes chargées, les opérations cryptographiques via l'API Android, les messages et les appels émis. Afin de suivre le rythme d'évolution d'Android, DroidBox instrumente le code des applications pour intercepter l'appel des fonctions d'intérêt. DroidBox intègre également une version de TaintDroid afin de détecter les éventuelles fuites d'information sensible. Son rapport intègre ainsi les fuites d'information via le réseau et par SMS que l'application aurait pu faire. Le lancement de l'application est automatique mais il appartient à l'utilisateur d'interagir avec l'application.

Andrubis [100], version Android d'Anubis, est un service d'analyse d'application en ligne. Il fournit les mêmes types d'information que DroidBox. Si DroidBox se limite aux fonctions de l'API Android, Andrubis lui considère le fait que les applications puissent également utiliser du code natif. Ils enregistrent ainsi dans leur rapport l'usage des appels système utilisés par l'application. Une autre différence par rapport à DroidBox est la génération d'évènements afin de couvrir tout le code. DroidBox se contente de lancer l'application en exécutant son `Activity` principal. Andrubis prend en compte le fait que chaque composant d'une application est un point d'entrée du programme et essaie de tous les lancer.

D'après l'étude effectuée par Neuner et al. dans [77], DroidBox aurait été utilisé comme base pour d'autres environnements d'analyse tels qu'ANANAS [44] et Mobile Sandbox [94]. Il existe également d'autres environnements indépendants de DroidBox tels que SmartDroid [111], CopperDroid [88], DroidScope [106] AASandbox [25] etc.

```
.method public onClick(Landroid/view/View;)V .locals 5
.parameter "v"
    .prologue const v4, 0x7f050004
    const/4 v3, 0x4
    const/4 v2, 0x0
    .line 92 iget-object v0, p0,
    Lcom/example/android/jetboy/JetBoy;->mJetBoyThread:
        Lcom/example/android/jetboy/JetBoyView$JetBoyThread;
    invoke-virtual {v0},
    Lcom/example/android/jetboy/JetBoyView$JetBoyThread;
        ->getGameState()I
    move-result v0
    const/4 v1, -0x1
    if-ne v0, v1, :cond_0
    .line 93 iget-object v0, p0,
    Lcom/example/android/jetboy/JetBoy;->
        mButton:Landroid/widget/Button;
    const-string v1, "PLAY!"
    invoke-virtual {v0, v1},
    Landroid/widget/Button;->setText(Ljava/lang/CharSequence;)V
    .line 94 iget-object v0, p0,
    Lcom/example/android/jetboy/JetBoy;->mTextView:
        Landroid/widget/TextView;
    invoke-virtual {v0, v2},
    Landroid/widget/TextView;->setVisibility(I)V
    .line 96 iget-object v0, p0,
    Lcom/example/android/jetboy/JetBoy;->mTextView:
        Landroid/widget/TextView;
    invoke-virtual {v0, v4},
    Landroid/widget/TextView;->setText(I)V
    .line 97 iget-object v0, p0,
    Lcom/example/android/jetboy/JetBoy;->mJetBoyThread:
        Lcom/example/android/jetboy/JetBoyView$JetBoyThread;
    invoke-virtual {v0, v2},
    Lcom/example/android/jetboy/JetBoyView$JetBoyThread;->
        setGameState(I)V
```

Listing 2.5 – Extrait du code `smali` de l'application JetBoy

```
1  public void onClick(android.view.View p6)
2      {
3          if(this.mJetBoyThread.getGameState() != 15) {
4              ...
5          } else {
6              this.mButton.setText("PLAY!");
7              this.mTextView.setVisibility(0);
8              this.mTextView.setText(1.76787404569e+38);
9              this.mJetBoyThread.setGameState(0);
10         }
11         return;
12     }
```

Listing 2.6 – Extrait du code résultant de la décompilation de l'application JetBoy avec le décompilateur par défaut d'Androguard

Nous avons présenté dans ce chapitre le système Android et introduit la base de connaissance nécessaire pour comprendre notre approche. Au lieu de détecter des scénarios d'attaque prédéfinis, nous souhaitons en découvrir de nouvelles et les détecter. La première étape à l'atteinte de cet objectif est d'analyser les flux d'information causés par les échantillons de malware Android. Pour observer ces flux, nous utilisons un moniteur de flux d'information Blare que nous avons porté sous Android. Dans le chapitre suivant, nous présentons ce moniteur et expliquons les ajouts effectués à Blare lors du portage sous Android.

Chapitre 3

Blare : un moniteur de flux d'information pour Linux et Android

Pour suivre les flux d'information, nous utilisons Blare [37], un système de détection d'intrusion paramètré par une politique de flux d'information. La politique de flux d'information décrit les flux autorisés dans le système et lève une larte à chaque violation de la politique. Blare est basé sur un modèle décrit en section 3.1 et est implémenté pour les systèmes de type Linux comme décrit en section 3.3. Android intégrant des fonctionnalités qui lui sont propres, le premier apport de la thèse a donc consisté à modifier Blare pour prendre en compte certaines fonctionnalités d'Android. La prise en compte de ces fonctionnalités donne une vue plus complète des flux d'information sous Android. La section 3.4 décrit les modifications apportées à Blare. Le résultat de ce portage, AndroBlare, nous sert d'environnement d'analyse dynamique d'application afin d'apprendre le comportement des applications à partir des flux d'information qu'elles causent dans le système. Dans la section 3.5, nous expliquons ainsi comment nous analysons les applications dans cet environnement.

3.1 Modèle théorique

Le modèle théorique de Blare [55, 56] décrit comment suivre et contrôler les flux d'information au niveau du système. Pour suivre les flux d'information, il utilise une méthode de marquage qui consiste à marquer les objets selon la nature de leur contenu. Il existe ainsi une distinction importante entre les conteneurs d'information et leur contenu.

Taille	8	16	32	64
Intervalle	$[-2^7, 2^7 - 1]$	$[-2^{15}, 2^{15} - 1]$	$[-2^{31}, 2^{31} - 1]$	$[-2^{63}, 2^{63} - 1]$

TABLE 3.1 – Intervalle de valeur du type `int` selon sa taille en bits

3.1.1 Conteneurs d'information

Blare suit les flux d'information au niveau du système d'exploitation. Il a ainsi une vue des flux entre les objets du système. Le modèle de Blare définit trois types de conteneur d'information : les conteneurs persistants, les conteneurs volatiles et les utilisateurs. Un conteneur persistant est un conteneur qui peut exister sur plusieurs cycles d'exécution du système. Un fichier est l'exemple typique d'un conteneur persistant. Un conteneur volatile est conteneur qui n'existe que le temps d'un cycle d'exécution du système. Il n'est créé qu'après le démarrage du système et est détruit à l'arrêt. Les sockets sont des exemples de conteneurs volatiles. Les utilisateurs sont les utilisateurs du système. À chacun de ces conteneurs sera associé un ensemble de marques ou tags qui permettent de suivre et contrôler les flux d'information. Nous noterons par la suite $\mathcal{C}$ l'ensemble des conteneurs d'information, $\mathcal{PC}$ l'ensemble des conteneurs persistants, $\mathcal{VC}$ l'ensemble des conteneurs volatiles et $\mathcal{U}$ l'ensemble des utilisateurs.

3.1.2 Information et contenu courant d'un objet : *itag*

Il existe deux types d'information dans le modèle de Blare. Le premier est l'information morte qui correspond à toute donnée non exécutée sur le système. Le contenu d'un fichier est l'exemple typique d'une information morte. Le deuxième est l'information exécutée qui correspond à toute donnée exécutée. Nous ne la retrouvons que dans les processus car seuls les processus peuvent exécuter des données. À chaque donnée dont nous souhaitons contrôler la propagation est associée un entier unique l'identifiant. L'ensemble des identifiants d'information morte est notée $I = \{1, 2, \ldots\}$ et correspond l'ensemble des entiers strictement positifs. L'ensemble des identifiants d'information exécutée est notée $X = \{-1, -2...\}$ et correspond à l'ensemble des entier strictement négatifs. En théorie, il est possible de suivre autant d'information que souhaité mais dans la pratique l'implémentation imposera toujours une valeur maximale et minimale des identifiants. Les identifiants sont représentés par des entiers signés de type `int` dans la version actuelle de Blare. La taille des types varie selon l'architecture du système (chapitre 11 de Linux Device Drivers [40]), ce qui rend impossible de définir une intervalle de valeur fixe pour les identifiants. Le tableau 3.1 liste les intervalles de valeur pour un `int` selon le nombre de bits utilisés pour le stocker. Il est à noter que le type `int` désigne un entier signé et le dernier bit de sa représentation en mémoire est ainsi utilisé pour coder son signe.

Si I et X sont deux ensembles distincts, il existe cependant une relation entre leurs éléments. Pour chaque élément n de I, $-n$ identifie la donné identifiée par n lorsqu'elle est exécutée dans un processus. Par exemple, si nous supposons que

le contenu du fichier `/system/bin/cat` [1] est identifié par 1 alors le processus qui exécutera `cat` contiendra l'information identifiée par -1.

Itag Le modèle de Blare marque chaque conteneur d'information par *tag* appelé *itag* pour caractériser son contenu. Sa valeur est un sous ensemble de $I \cup X$ et pour tout conteneur c, $itag(c) = \{i_1, \dots i_n\}$ signigie que le contenu de c contient les informations initialement identifiées par $i_1, \dots i_n$.

$$\forall c \in \mathcal{C},\ itag(c) \in \mathscr{P}(I \cup X)$$

3.1.3 Politique de flux d'information : $(x)ptag$

Blare est paramétré par une politique de flux d'information. Cette politique définit les flux autorisés entre les conteneurs d'information et est représentée par l'ensemble des *ptags* dans le système. À chaque conteneur d'information est ainsi associé un deuxième tag : *ptag*. Le *ptag* définit les différents contenus maximaux autorisés pour le conteneur. Sa valeur est une collection de sous-ensemble de $I \cup X$ et chacun de ses éléments représente un contenu maximal légal pour le conteneur.

$$\forall c \in \mathcal{C},\ ptag(c) \in \mathscr{P}(\mathscr{P}(I \cup X))$$

Le contenu courant d'un conteneur c est légal si $itag(c)$ est inclus dans au moins un des éléments de $ptag(c)$. Le *ptag* d'un conteneur est nul par défaut. Le conteneur n'a donc pas le droit d'accéder à une donnée à laquelle un identifiant a été associé. Toute information n'ayant aucun identifiant associé est considéré comme non sensible et a le droit de se propager dans n'importe quel conteneur.

$$\text{Le contenu de } c \text{ est légal si } \exists e \in ptag(c) \text{ tel que } itag(c) \subseteq e$$

Par exemple, si le *ptag* d'un conteneur vaut $\{\{1,2\},\{2,3\}\}$ cela signifie que l'ensemble des contenus légaux possibles pour ce conteneur sont $\varnothing$, $\{1\}$, $\{2\}$, $\{3\}$, $\{1,2\}$ et $\{2,3\}$.

Politique d'un programme, politique d'un processus : $xptag$

Une distinction importante est faite entre programme et processus. Toute donnée pouvant être exécutée dans le système est un programme tandis qu'un processus est une instance d'un programme ou plus précisément une entité exécutant un programme. Le contenu d'un fichier `com.android.browser.dex` est un exemple de programme Android et le processus `com.android.browser` est une instance de ce programme.

1. `cat` est une application en ligne de commande qui concatène et affiche le contenu de fichiers

```
USER      PID   PPID   NAME
root      1     0     /init
drm       179   1     /system/bin/drmserver
media     180   1     /system/bin/mediaserver
install   181   1     /system/bin/installd
keystore  183   1     /system/bin/keystore
radio     186   1     /system/bin/netmgrd
nobody    187   1     /system/bin/sensors.qcom
root      190   1     /system/bin/thermal-engine-hh
root      1678  1     /system/xbin/su
```

FIGURE 3.1 – Extrait de la sortie de la commande `ps` sous Android 4.4

xptag La politique d'un programme est exprimée par un troisième tag appelé *xptag* qui est attaché au conteneur stockant le programme. Tout comme le *ptag*, le *xptag* est également une collection de sous-ensemble de $I \cup X$. Il n'est pas utilisé pour vérifier la légalité d'un contenu mais pour calculer la valeur du *ptag* des processus.

$$\forall c \in \mathcal{C},\ xptag(c) \in \mathscr{P}(\mathscr{P}(I \cup X))$$

Un processus est une instance d'un programme à l'exécution. Dans les systèmes Unix/Linux, les processus peuvent être listés avec la commande `ps`. La figure 3.1 présente un extrait de la sortie de la commande `ps` sous Android. Les informations d'intérêt sont celles de la première, la deuxième et la dernière colonnes. Chaque ligne liste des informations sur un processus. La première colonne identifie l'utilisateur exécutant le processus. La deuxième colonne liste les identifiants des processus. Enfin la dernière colonne liste les programmes exécutés par les processus. Ainsi, le programme `/init` est exécutée par l'utilisateur `root` dans le processus identifié par 1.

Un processus étant le résultat de l'exécution d'un programme par un utilisateur, sa politique est calculée à partir de celle de l'utilisateur et celle du programme qu'il exécute. Nous noterons $exec(c, u)$ le processus résultant de l'exécution du programme c par l'utilisateur u par la suite. Plus précisément, la politique d'un processus résulte de l'intersection de la politique de l'utilisateur et de celle du programme qu'il souhaite exécuter. Soient un programme c et un utilisateur u. La politique associée au processus $exec(c, u)$ est l'intersection des éléments de la politique de c avec ceux de la politique de u.

$$\forall p \in Prog \text{ et } \forall u \in U,\ ptag(exec(c,u)) = \{e_c \cap e_u\}, e_c \in ptag(c) \text{ et } e_u \in ptag(u)\}$$

Parmi les processus listés par la commande `ps` dans la figure 3.1 se trouve un processus exécuté par l'utilisateur `media` et exécutant le programme dans le fi-

chier `/system/bin/mediaserver`. En supposant que la politique associée respectivement au programme et à l'utilisateur soit $\{\{1,2,3\},\{3,4\}\}$ et $\{\{2,3,5\},\{5,12\}\}$, alors la politique associée au processus serait $\{\{2,3\}\}$. Cet unique élément de la politique résulte de l'intersection du premier élément de la politique du programme avec celui de l'utilisateur. Les autres intersections sont toutes vides. Cela signifie que le processus peut contenir tout au plus les informations identifiées par 2 et 3.

3.1.4 Suivi et contrôle de flux d'information

Le contenu courant d'un objet et sa politique de flux sont représentés respectivement par l'*itag* et le *ptag* attachés à l'objet. L'*itag* caractérisant le contenu courant d'un objet, sa valeur évoluera en fonction de l'évolution du contenu de l'objet auquel il est attaché. Le *ptag* par contre lui n'évoluera que si la politique de flux d'information n'évolue. Le suivi et le contrôle de flux d'information se fait en deux étapes. La première étape consiste à la mise à jour de l'*itag* de l'objet dont le contenu a été modifié suite au flux observé. La deuxième étape consiste à vérifier la légalité du flux observé. Un flux est légal si le contenu des objets modifiés par le flux est légal. La vérification de la légalité se fait en comparant la nouvelle valeur de l'*itag* de l'objet modifié suite au flux avec la politique associée à l'objet, à savoir son *ptag*.

3.1.4.1 Suivi de flux d'information

Blare observe les évènements du système provoquant des flux d'information et à chaque flux observé, il met à jour l'*itag* des objets dont le contenu a été modifié par le flux. Ces flux peuvent être classés dans différents groupes comme listé dans le tableau 3.2. Chaque ligne décrit l'interaction entre deux objets du système causant un flux d'information et la manière dont les tags sont mis à jours quand Blare observe le flux. Non seulement l'*itag* de l'objet modifié est mis à jour mais aussi son *xptag*, sauf dans le cas de l'exécution d'un programme.

Le *xptag* représente la politique d'un programme. Or un programme est une donnée stockée dans un conteneur. Un programme peut ainsi être propagé dans le système lors des flux d'information. Afin de s'assurer que la politique d'un programme reste associée à ce programme malgré le fait qu'il puisse être copié dans d'autres conteneurs, Blare propage le *xptag* des fichiers en même temps que l'*itag* du fichier.

À chaque fois Blare observe un flux d'information, il effectue une suraproximation du flux observé. Il considère le pire des cas qui est que le contenu entier de la source du flux d'information s'est propagé vers le conteneur destination. Nous expliquons ci-dessous ces mises à jour effectuées par Blare listées dans le tableau 3.2.

Lorsqu'un processus p lit le contenu d'un conteneur f, $Read(p, f)$, Blare considère que le contenu de f s'est propagé dans p. Le contenu du processus est modifié par le flux et son *itag* change de valeur. Le pire des cas étant que tout contenu de f se soit propagé dans p, Blare considère alors que le nouveau

Type d'évènement	Description	Mise à jour des tags
$Read(p, f)$	Lecture de f par p	$itag(p) \leftarrow itag(p) \cup itag(f)$ $xptag(p) \leftarrow xptag(f) \cap xptag(p)$
$Write(p, f)$	Écriture de f par p	$itag(f) \leftarrow itag(p) \backslash X$ $xptag(f) \leftarrow xptag(p)$
$Write_{app}(p, f)$	Écriture de f par p	$itag(f) \leftarrow (itag(p) \backslash X) \cup itag(f)$ $xptag(f) \leftarrow xptag(f) \cap xptag(p)$
$Exec(p, f)$	Exécution du contenu de f par p	$itag(p) \leftarrow \{-n\},\ n \in itag(f)$ $ptag(p) \leftarrow xptag(f)$

TABLE 3.2 – Blare : méthode de suivi de flux

contenu de p est un mélange de son ancien contenu et du contenu de f. La nouvelle valeur de l'*itag* de p est donc l'union de son ancien *itag* avec celui de f. L'exemple typique de ce type de flux est la lecture d'un fichier par un processus. Les fichiers ne sont cependant pas les seuls conteneurs d'information du système à partir duquel un processus peut lire des données. Un processus peut également lire des données à partir d'un socket ou d'un message via les files de message [65]. Le conteneur f ne représente donc pas uniquement un fichier mais tout autre conteneur à partir duquel un processus peut lire des données.

Lorsqu'un processus p écrit dans un conteneur f, $Write(p, f)$, Blare considère que les informations dans p se propagent vers le conteneur f. Le contenu de f change et Blare met à jour son *itag*. Le pire des cas étant que tout le contenu de p se propage dans f, le nouveau contenu de f est en conséquent un mélange de son ancien contenu avec le contenu de p à l'exception des données exécutées. Blare considère qu'un processus ne propage pas le code qu'il exécute. Les identifiants négatifs ne sont donc pas propagés dans l'*itag* du destinataire. La nouvelle valeur de l'*itag* de f est ainsi l'union de son ancienne valeur avec l'ensemble des identifiants positifs dans l'*itag* de p.

Comme expliqué en section 3.1.2, les données sensibles exécutées dans un processus sont identifiées par un entier strictement négatif. Lorsqu'un processus exécute un programme, Blare considère que son contenu courant est remplacé par le code provenant du fichier contenant le programme. Son *itag* devient ainsi l'ensemble des valeurs négatives des identifiants présents dans l'*itag* du fichier.

Les interactions entre les objets d'un système ne se limite à la lecture, écriture et l'exécution. D'autres évènements tels que la création de conteneur ou l'envoi de signaux existent également. Ces évènements n'engendrent pas de flux d'information explicites entre les objets et sont ignorés par Blare. Il est cependant techniquement possible de se servir de ces évènements pour faire fuir des informations et ainsi échapper au contrôle effectué par Blare.

3.1.4.2 Contrôle de flux d'information

Un flux d'information est légal si le contenu de l'objet qui a été modifié par le flux est légal. Le contrôle d'un flux d'information se fait en deux étapes.

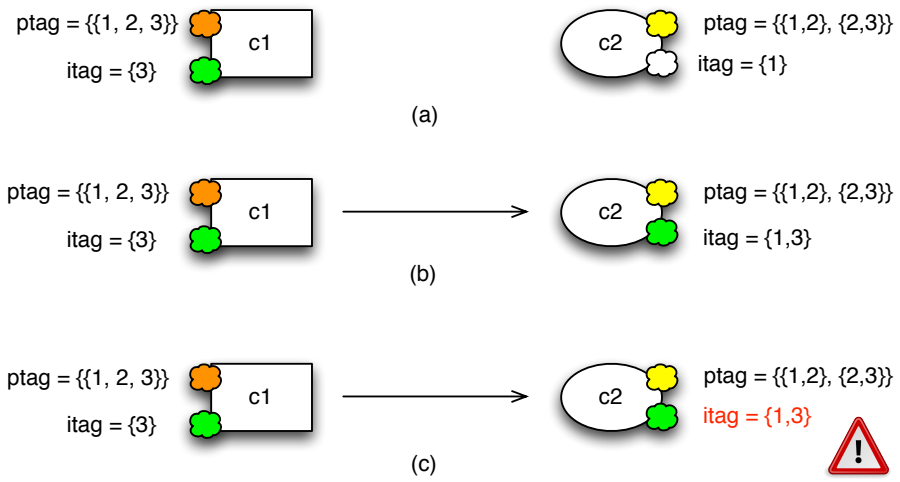


FIGURE 3.2 – Exemple de contrôle de flux d'information effectué par Blare

1. Mise à jour de l'*itag* de l'objet dont le contenu a été changé.

2. Contrôle de la légalité du nouveau contenu de l'objet modifié par le flux. Si le contenu est illégal alors, Blare considère que le flux observé est illégal et lève une alerte.

La figure 3.2 illustre un exemple de contrôle de flux lorsque Blare observe la lecture d'un fichier $c1$ par un processus $c2$. (a) Les deux conteneurs sont marqués préalablement avant la lecture. (b) Lors de la lecture du fichier, Blare considère que le contenu de $c1$ se propage vers $c2$. Il met ainsi à jour la valeur de $itag(c2)$ comme décrit dans le tableau 3.2 pour la lecture de fichier. La valeur de $itag(c2)$ devient $\{2, 3\}$. (c) Après la mise à jour de l'*itag*, Blare contrôle la légalité du nouveau contenu afin de déterminer si le flux observé est légal ou non. Il n'existe pas d'élément de $ptag(c2)$ qui inclut $itag(c2)$. Le nouveau contenu est illégal, ce qui signifie que le flux observé est illégal. Une alerte est donc levée.

Blare est outil de détection d'intrusion paramétré par une politique de flux d'information. Pour détecter une intrusion, il faut ainsi dans un premier temps définir une politique de sécurité qui identifie les informations à protéger et où ces informations ont le droit de se propager. Dans la section suivante, nous présentons une manière de définir cette politique de flux sous Android.

3.2 Définition d'une politique de flux d'information

Protéger un système avec un outil tel que Blare signifie qu'il faut dans un premier temps définir une politique de sécurité. Dans le cas de Blare, cette politique est une politique de flux d'information. Elle identifie les informations à protéger et comment les protéger, c'est-à-dire la liste des conteneurs où elles ont le droit de se propager. Définir une politique de flux d'information pour Blare se résume en quatre étapes.

1. Identifier les informations sensibles à protéger.
2. Associer à chacune de ces informations un identifiant unique.
3. Identifier les conteneurs légaux pour ces informations.
4. Définir la valeur des *ptags* à associer à ces conteneurs.

Il existe deux approches pour définir la politique de sécurité. La première consiste à la calculer à partir d'une autre politique existante telle qu'une politique de contrôle d'accès. Dans [56], Geller et al. ont proposé un algorithme qui transforme une politique de contrôle d'accès mandataire App Armor en une politique de flux d'information Blare. Les politiques App Armor définissent des profils de programme limitant leur accès aux autres objets du système tels que les fichiers. L'algorithme proposé prend en entrée un ensemble de profils App Armor et donne en sortie leur équivalent en terme de politique Blare. Cette approche a le mérite d'être automatique et rapide mais elle ne permet pas d'exploiter toute l'expressivité d'une politique Blare. Un exemple de ce manque d'expressivité est la restriction sur les mélanges d'information. Un conteneur a le droit d'accéder aux informations i et j mais n'a pas le droit de les mélanger, c'est-à-dire que le conteneur contient soit i soit j mais pas les deux en même temps. Le listing 3.1 présente une politique App Armor pour le programme `apache`. Cette politique autorise le programme à lire le contenu des fichiers `apache2.conf` et `index.php`. La politique de flux qui peut être déduite de cette politique est que les données dans les deux fichiers ont tout deux le droit de se propager vers le programme `apache`. Cependant, le propriétaire du contenu de ces deux fichiers peut souhaiter que le contenu de ces fichiers ne se mélangent pas. Cette contrainte ne peut-être exprimée avec une politique App Armor et une politique de flux exprimant une telle restriction ne peut donc être calculée à partir d'une politique App Armor.

```
{/usr/bin/apache,
        {(/etc/apache2.conf, w),
        (/etc/apache2.conf, r),
        (/www/index.php,r), (/usr/bin/ftpd, px)}
}
```

Listing 3.1 – Exemple de politique App Armor

La deuxième approche consiste à définir manuellement la politique de flux d'information. Cette approche est plus longue car elle implique une analyse complète du système mais permet d'exploiter toute l'expressivité d'une politique Blare.

Construction manuelle d'une politique pour Android

Dans [16], nous avons proposé une politique de flux d'information pour le système Android. La politique a été construite à la main en se basant sur une analyse manuelle du système. Le but était de fournir une politique à taille réelle pour un système entier. Comme écrit précédemment, la définition d'une politique de flux se fait en quatre étapes.

La première étape est l'identification des informations sensibles à surveiller. Nous avons construit la liste des informations sensibles à partir des données que nous avons jugées sensibles sur le système Android. Ces informations incluent les données liées à l'utilisateur (ex : liste de contact et données de géolocalisation) mais aussi celles liées aux applications (ex : code et paramètres de configuration). Les données sensibles ont été identifiées en analysant manuellement le contenu des fichiers sur le système et en prenant en compte celles qui sont protégées par le mécanisme de permission. Notre politique recense 150 informations sensibles mélangeant données non exécutées et code exécuté à l'intérieur des processus. Il existe encore plus d'information sur un système Android [2] mais nous avons jugé cette quantité d'information à surveiller suffisante pour construire une politique de flux d'information pour tout un système.

La deuxième étape consiste à identifier les conteneurs d'information. La liste des fichiers et programmes installées sur un système Android est assez simple à construire. Grâce à la commande `find`, nous pouvons lister tous les fichiers du système. Quant aux applications, les applications livrées sous forme d'`apk` sont connues grâce à la commande `pm` et les applications natives sont stockées dans `/system/bin` et `/system/xbin`. La liste des utilisateurs est elle par contre moins évidente à construire car contrairement à un système Linux standard, il n'existe pas de fichier `/etc/passwd` ni de fichier `/etc/shadow`. Android définit en dur un ensemble restreint d'utilisateur nécessaire au fonctionnement de base du système tels que `root`, `radio` et `system`. Puis pour chaque application non native installée sur le système, il associe un utilisateur unique. Pour établir la liste des utilisateurs associés aux applications, nous nous sommes basés sur le contenu des fichiers `/data/system/packages.list` ou `/data/system/packages.xml`. Ils recensent les informations liées aux applications installées sur le téléphone dont l'utilisateur qui leur associé.

Une fois les données sensibles et conteneur identifiés, nous avons ensuite construit la politique de flux. La politique de chaque programme a été construite en analysant ses sources pour comprendre à quelles informations il accède. La politique de chaque utilisateur a été obtenue en faisant l'union des politiques des programmes qu'il peut exécuter. Dans la plupart des cas, il n'y aura qu'un

2. Il existe plus des milliers de fichiers réguliers lors de l'exécution du système.

	Identifiant des informations									
	11	-44	10	6	-55	-39	3	9	37	-37
Fichiers										
calendar.db	×									
contacts2.db			×	×						
mms.db							×			
telephony.db								×		
TelephonyProvider.apk									×	
Programmes										
com.android.calendar	×	×								
com.android.providers.contact			×	×	×					
com.android.providers.telephony							×	×		×
Utilisateurs										
10024	×	×								
10004			×	×	×	×				
1001							×	×		×

TABLE 3.3 – Un extrait de la politique de flux d'information

seul programme car en dehors des utilisateurs système les autres utilisateurs n'exécuteront que l'application à laquelle ils ont été associés. La politique des fichiers a été construite à partir des données qu'elles contiennent lors de l'analyse et de la politique des programmes qui y accèdent. Si une donnée sensible a été trouvée dans un fichier, nous avons supposé que ce fichier avait le droit de contenir la donnée. À ces données s'ajoutent celles auxquelles les applications qui ont le droit d'écriture au fichier peuvent accéder. Nous avons supposé que toute application avait le droit de stocker les données contenues dans les fichiers auxquels l'application a un accès légal. Généralement, il s'agit des fichiers dans le répertoire local de l'application. Pour les 150 informations identifiées lors de la première étape, nous avons identifié 186 conteneurs d'information pouvant accéder à ces informations où les stocker. La politique contient donc 186 *ptags*.

La politique finale a été représentée sous la forme d'une matrice dont un extrait est présentée dans le tableau 3.3. Le chemin complet des fichiers a été omis pour une meilleure lisibilité. Une croix à l'intersection d'une ligne et d'une colonne signifie que le conteneur correspondant à la ligne a le droit de stocker l'information correspondant à la ligne. L'ensemble des croix sur une ligne l détermine ainsi le contenu maximal autorisé pour le conteneur correspondant à l. L'utilisateur 10004 (avant dernière ligne) a par exemple le droit d'accéder aux informations identifiées par 6, 10, -39 et -55. Autrement dit, son *ptag* vaut $\{\{-55, -39, 6, 10\}\}$.

Si la pertinence de la politique n'a pas été évaluée dans ce travail, le travail effectué a cependant permis de mettre en évidence deux points. Le premier est

la difficulté de la création de la politique à la main car elle a nécessite une analyse complète du système pour identifier les informations sensibles et leurs conteneurs légaux. Le deuxième point est le nombre d'information sensible sur le système. Contrairement à ce que les permissions Android pourraient laisser croire, il peut y avoir un nombre assez conséquent d'information sensible sur le système. La politique que nous avons construite recense 150 informations mais il y a encore plus d'information que cela dans un système.

3.3 KBlare : suivi et contrôle de flux via LSM

Le cœur de Blare a été développée en tant que module de sécurité Linux. Il réside dans le noyau et est appelé KBlare. Dans le reste du document, nous désignons par KBlare l'implémentation de Blare dans le noyau et Blare tout l'environnement d'exécution Blare (KBlare plus les outils en espace utilisateur lié à Blare). Les flux d'information au niveau du système se font via les interactions entre les objets du système et plus précisément via les appels système effectués par les processus. Pour contrôler les flux d'information, KBlare intercepte ainsi les appels système en utilisant les *hooks* LSM [103, 102]. LSM est un *framework* de sécurité dans le noyau utilisé pour implémenter les modules de sécurité Linux. Les *hooks* sont des appels aux modules de sécurité du noyau. Ils sont placés dans le flux d'exécution des appels système afin que les modules de sécurité puisse intercepter et contrôler ces appels. Pour intercepter un appel système, un module de sécurité enregistre une fonction à associer au *hook* correspondant à l'appel système. À chaque fois que ce *hook* est atteint, le noyau appelle la fonction enregistrée afin que le module de sécurité effectue le contrôle souhaité.

La figure 3.3 illustre le fonctionnement général de LSM. Lors d'un appel système, le noyau vérifie si l'accès demandé est autorisé pour le processus effectuant l'appel. Il commence par vérifier la conformité avec la politique DAC du système. Ensuite, il donne la main à LSM qui lui demande au module de sécurité actif[3] s'il autorise ou non l'accès demandé. Cette étape correspond au moment où un *hook* est atteint dans le flux de traitement de l'appel système. La fonction enregistrée au *hook* est ainsi appelée et statue si l'accès est autorisé. Lorsqu'un processus fait un appel système, KBlare intercepte l'appel, calcule le flux correspondant, met à jour les tags des objets modifiés et contrôle la légalité du flux observé. KBlare se contente cependant de détecter les intrusions mais ne les prévient pas. Même en cas de violation de la politique de flux d'information, il autorise le noyau à continuer le traitement de l'appel système.

Le tableau 3.4 liste les *hooks* utilisés par KBlare dont l'opération correspondante peut engendrer des flux d'information. La première colonne liste les *hooks*, le deuxième les opérations interceptées via le *hook* et la dernière colonne les types de flux que KBlare peut observer. D'autres *hooks* sont utilisés pour le fonctionnement interne de KBlare mais les opérations correspondantes n'induisent aucun flux d'information.

3. En général un seul module LSM est actif

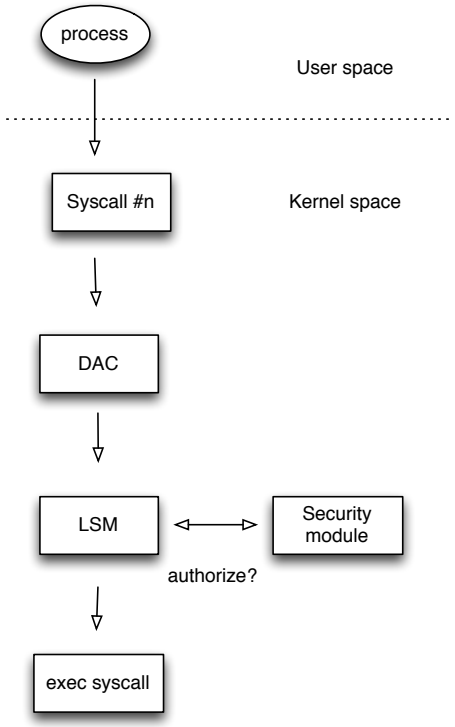


FIGURE 3.3 – Design de Linux Security Module

Hook	Opération	Flux possible observé
`file_permission`	Lecture/écriture à un fichier ouvert	Entre le processus et le fichier accédé
`file_mmap`	Chargement en mémoire du contenu d'un fichier	Du fichier vers le processus
`inode_permission`	Accès à un i-node	Entre un pipe et un processus
`bprm_set_creds`	Exécution d'une application native	Du fichier contenant l'application vers le processus
`socket_sock_rcv_skb`	Réception d'un paquet réseau	De paquet reçu vers le processus
`socket_sendmsg`	Envoi d'un message dans un socket réseau	Du processus vers le socket et le paquet
`socket_recvmsg`	Réception d'un message via un socket	Du socket vers le processus
`unix_may_send`	Envoi d'un message via un socket UNIX	Du processus vers le socket UNIX
`msg_queue_msgrcv`	Réception d'un message via une file de message	Du processus vers le message envoyé
`msg_queue_msgsnd`	Envoi d'un message dans une file de message	Du message reçu vers le processus destinataire du message

TABLE 3.4 – Liste des *hooks* LSM utilisés par KBlare pouvant engendrer un flux d'information

Stockage des tags

KBlare représente les tags des objets du système avec des structures contenant l'*itag*, le *ptag* et le *xptag* associés aux objets. En dehors des *hooks*, LSM a également modifié les structures représentant les objets du système pour ajouter un champ appelé `security`. Ce champ est un pointeur générique destiné à être utilisé par les modules de sécurité. KBlare utilise ce champ pour stocker le pointeur vers la structure représentant les tags.

Les structures représentant l'ensemble des tags existent uniquement en mémoire et n'existent donc que le temps d'un cycle d'exécution du système. Afin de stocker de manière persistante les tags associés aux fichiers, KBlare stocke également leurs tags dans les attributs étendus des fichiers. Les attributs étendus sont des zone de mémoire sur le disque qui sont utilisées par les systèmes de fichier pour stocker des méta-données liées aux fichiers.

KBlare contrôle les flux d'information en interceptant les appels système et pour intercepter les appels système il utilise les *hooks* LSM. Ces appels permettent les interactions entre les objets du système : accès aux fichiers, exécution de code natif et communication entre processus. Le noyau d'Android est basé sur celui de Linux mais contient des fonctionnalités en plus. Dans la section suivante, nous présentons comment nous avons modifié KBlare afin de prendre en compte les fonctionnalités spécifiques à Android.

3.4 AndroBlare : Blare sous Android

Le premier apport de cette thèse est le portage de Blare sous Android. Nous avons montré précédemment que Blare contrôlait les flux d'information traduisant les accès aux fichiers, les exécution de code et les communications entre processus. Ces contrôles sont basés uniquement sur les interactions possibles entre les objets du système sous Linux. Or, Android intègre des fonctionnalités qui lui sont propres pouvant engendrer des flux d'information. Ces fonctionnalités sont l'exécution des applications par les machines virtuelles Dalvik et les communications via le Binder.

Le Binder est un mécanisme de communication implémenté principalement dans le noyau Android. Il est la base des communications entre processus proposés par le framework Android tels que les `intents`. Android n'utilise pas les mécanismes de communication offerts par Linux. Les communications entre applications étaient donc invisibles à KBlare.

Les applications que nous avons désignées par la couche *Applications* sur la figure 2.1 sont livrées sous la forme de *dalvik bytecode* et interprétées par la machine virtuelle Dalvik. Elles ne sont donc pas exécutées de manière native. À leur *exécution*, le processus contenant la machine virtuelle qui va interpréter le code n'effectue pas d'appel système `execve` mais se contente de lire puis charger le code de l'application en mémoire. Aucun appel système traduisant l'exécution d'une application n'ayant lieu, KBlare ne voyait donc pas l'*exécution* des applications. En conséquent, aucune politique ne pouvait être appliquée aux applications Android et aucun processus n'était marqué comme exécutant une application lorsque le code de cette application était associée à un identifiant.

Limitations de Blare
(1) Il n'y avait aucune visibilité sur les communications entre applications Android.
(2) KBlare ne détectait pas l'exécution des applications Android. Les tags des processus n'indiquaient jamais l'exécution d'une application dont le code est sensible et aucune politique ne pouvait être appliquée aux applications.

Dans ce qui suit, nous présentons comment nous avons modifié Blare pour combler ce manque de visibilité au niveau des flux d'information sous Android.

Nous présentons le fonctionnement du Binder puis comment nous avons implémenté le suivi des flux d'information via ce mécanisme. Ensuite, nous expliquons comment les applications Android sont `exécutées` et comment nous avons ajouté a prise en compte de leur *exécution* dans Blare.

3.4.1 Communication entre processus : Binder

3.4.1.1 Fonctionnement

Les applications Android tournent dans des processus distincts mais communiquent souvent entre elles grâce à divers mécanismes IPC tels que les intents et les Content Providers. Ces mécanismes sont fournis par le *framework* Java Android mais se basent tous à un niveau plus bas sur le Binder. Le Binder est un mécanisme IPC qui permet à une application d'appeler une méthode distante, c'est-à-dire implémentée au sein d'une autre application tournant dans un autre processus. Le cœur de ce mécanisme est implémenté dans le noyau sous forme de *driver*[4] dans le noyau. Par la suite nous adopterons les termes suivants. Le Binder fait référence au code du Binder dans le noyau. Un service est une application dont les méthodes peuvent être appelées par d'autres applications. Un client est une application qui appelle une méthode d'un service.

La figure 3.4 illustre le fonctionnement du binder lors de l'appel d'une méthode distante. Tout client doit avoir une référence du service implémentant la méthode qu'il souhaite invoquer. Le client commence ainsi par demander la référence du service ciblé à un autre service appelé Context. Context, implémenté sous Android par le Service Manager, est un service d'annuaire qui fait correspondre à chaque nom de service une référence unique. Chaque service doit s'enregistrer auprès du Service Manager afin que ses méthodes puissent être invoquées par d'autres applications. Lorsqu'un client souhaite obtenir la référence d'un service, il appelle ainsi la méthode `getService` du Service Manager en lui donnant en argument le nom du service. Une fois la référence du service obtenu, le client appelle la méthode. L'invocation de la méthode n'est pas directe. Le client ne communique pas directement avec le serveur mais transmet l'appel au binder qui lui le transmet au serveur. Une fois l'appel reçu puis exécuté par le serveur, ce dernier renvoie le résultat au binder qui lui le transmet au client.

3.4.1.2 Implémentation du Binder dans le noyau

Techniquement parlant, le Binder est implémenté sous forme de driver dans le noyau. Cela signifie qu'il est présenté dans l'espace utilisateur en tant que fichier, `/dev/binder`, et que les seules interactions possibles sont des appels système sur ce fichier. Le listing 3.2 liste les opérations sur les fichiers, voir le chapitre 3 de [40], supportés par Binder. Elles sont listés dans une structure appelée `file_operations` qui associe à chaque opération supportée la fonction correspondante. Par exemple, la valeur du champ `open` correspond à la fonc-

4. `<path-to-kernel>/drivers/staging/android/binder.c`

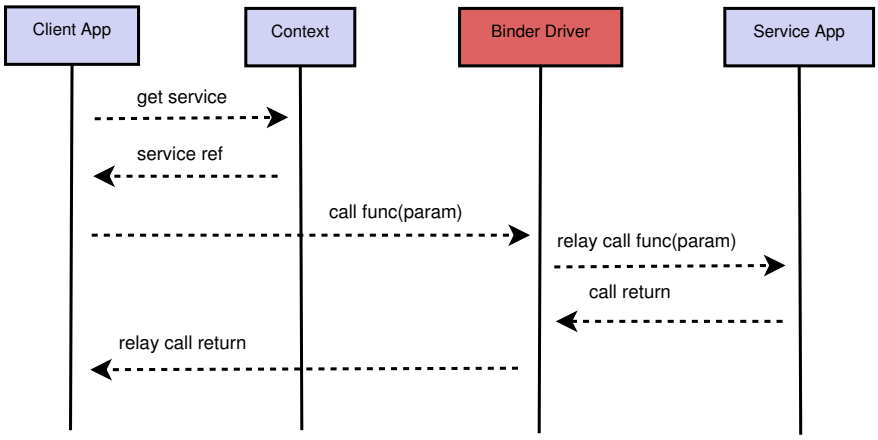


FIGURE 3.4 – Appel d'une méthode distante grâce au Binder

tion, ici `binder_open`, qui sera appelée à chaque fois qu'un processus ouvre `/dev/binder`.

Celle qui a intérêt pour nous est la fonction `binder_ioctl` (listing 3.3). Elle est appelée quand un processus effectue un appel système `ioctl` et prend les arguments de l'appel système. Le deuxième argument de la fonction, `cmd`, correspond à la commande à exécuter par le Binder. Binder reconnaît 5 commandes.

- `BINDER_WRITE_READ` est la commande la plus importante. Elle sert à l'envoi et réception de données.
- `BINDER_SET_MAX_THREADS` définit le nombre maximal de threads par processus pour traiter les messages reçus via le Binder.
- `BINDER_SET_CONTEXT_MGR` définit le `Context Manager`. Une fois défini, il ne peut être modifié.
- `BINDER_THREAD_EXIT` notifie l'arrêt d'un thread utilisé pour le traitement des messages reçus via le Binder.
- `BINDER_VERSION` retourne le numéro de version du Binder.

Lorsqu'un processus envoie la commande `BINDER_WRITE_READ` dans le but d'envoyer des données, il définit une deuxième commande précisant le type d'envoi. Il existe 14 commandes[5] liées à l'envoi de données mais seules deux d'entre elles correspondent à l'envoi des transactions : `BINDER_TRANSACTION` et `BINDER_REPLY`. `BINDER_TRANSACTION` correspond à l'émission d'une transaction d'un processus client appelant une méthode distante. `BINDER_REPLY` correspond à l'envoi du résultat de l'appel par le processus ayant exécuté la fonction. L'envoi d'un message ayant un de ces deux types initie l'appel à la fonction `binder_transaction` qui implémente la troisième et la cinquième étapes de l'appel d'une méthode distante comme décrit sur la figure 3.4 : `call func(param)` et `call return`. Toute action exécutée par le noyau se fait dans le contexte

5. Voir la fonction `binder_thread_write`

d'un processus à l'exception des interruptions [6]. Dans le cas des appels système, le contexte d'exécution est celui du processus ayant fait l'appel ce qui signifie que l'envoi des transactions se fait dans le contexte de l'émetteur.

Lorsque le Binder transmet une transaction à un processus, il l'ajoute à une liste de tâche à traiter par ce processus. Les processus en attente de transaction consultent ainsi cette liste de tâche pour traiter les transactions qui lui ont été transmises grâce à la commande `BINDER_WRITE_READ`. Chacune des tâches a un type défini. Il existe 6 types de tâche en tout dont `BINDER_WORK_TRANSACTION` qui correspond au traitement d'une transaction en attente. À la réception d'une tâche de ce type, les informations liées à la transaction sont envoyées au processus en espace utilisateur afin qu'il puisse exécuter l'appel.

Le Binder est le cœur des IPC sous Android mais son usage n'était pas contrôlé par KBlare. Il y a deux raisons à cela. La première est que son usage se fait via l'appel système `ioctl` or KBlare n'utilise pas le *hook* permettant d'intercepter cet appel. La deuxième raison est que Binder est un mécanisme propre à Android. Aucun travail ne s'est donc intéressé à étendre LSM afin de pouvoir contrôler l'usage du Binder au moment où nous nous sommes intéressés au Binder durant la thèse. À cause de ces raisons, KBlare ignorait les flux d'information via le Binder et ne pouvait observer les communications entre applications Android. Nous présentons dans la section qui suit comment KBlare a été étendu pour suivre les flux d'information causés par son usage.

```
1  static const struct file_operations binder_fops =
2  {
3          .owner = THIS_MODULE,
4          .poll = binder_poll,
5          .unlocked_ioctl = binder_ioctl,
6          .mmap = binder_mmap,
7          .open = binder_open,
8          .flush = binder_flush,
9          .release = binder_release,
10 };
```

Listing 3.2 – Liste des opérations sur les fichiers supportées par Binder

```
1  static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg);
```

Listing 3.3 – Signature de la fonction `binder_ioctl`

6. Ce n'est jamais notre cas dans le Binder

3.4.2 Suivi de flux d'information dans le Binder

Pour suivre les flux d'information liées aux transactions via le Binder, deux solutions ont été envisagées. La première solution est d'intercepter l'appel système `ioctl` grâce au *hook* `file_ioctl`. Cette solution est limitée car elle ne permettrait de voir que l'interaction entre les processus et `/dev/binder`. Or nous avons expliqué que le véritable flux s'opérait entre processus et non entre un processus et `/dev/binder`. Cette limitation pourrait cependant être contourné en implémentant tout le mécanisme de traitement des commandes envoyées via `ioctl` au Binder mais cette approche est assez lourde.

La deuxième solution, celle que nous avons retenu, consiste à étendre LSM afin de permettre aux modules de sécurité de contrôler l'envoi et la réception des transactions via le Binder. L'extension de LSM que nous avons apporté consiste à rajouter un champ `security` à la structure représentant les transactions et deux nouveaux *hooks* pour intercepter l'envoi et la réception de transaction. Tout comme les messages envoyés dans les files de message, nous considérons que les transactions sont des conteneurs volatiles. Nous avons ainsi ajouté un champ générique `security` à la structure les représentant. KBlare utilisera ensuite ce champ pour référencer les tags associés à la transaction. Les deux nouveaux *hooks* ont été rajoutés dans le code de Binder [7]. Quant aux *hooks*, nous les avons ajouté dans le corps du Binder pour intercepter les échanges de transaction. Plus précisément, nous avons ajouté un `hook` pour intercepter l'envoi d'une transaction et une autre pour la réception. Sous Android, KBlare utilise ces *hooks* pour suivre et contrôler les flux d'information [8] via le Binder.

Interception de l'envoi d'une transaction binder_transaction_write

Nous avons inséré ce *hook* dans le corps de la fonction `binder_transaction`. Cette fonction est utilisée par Binder pour transmettre les transactions émises par les processus. C'est dans cette fonction que la nouvelle tâche correspondant à la transaction est ajoutée à la liste de tâche en attente du destinataire. Le *hook* est inséré avant l'ajout de la nouvelle tâche.

Dans KBlare, nous avons déclaré une fonction correspondant à ce *hook* qui propage les tags du processus vers la transaction. La fonction crée une instance de la structure contenant le tag des conteneurs d'information, l'associe à la transaction et copie les tags du processus (*itag* et *xptag*) vers la structure contenant les tags de la transaction. Nous considérons que les transactions ont le droit de contenir n'importe quelle information. Aucun contrôle n'est ainsi effectué.

Interception de la réception d'une transaction binder_thread_read

Le deuxième *hook* est inséré dans le corps de la fonction `binder_thread_read`. Cette fonction est utilisée pour récupérer et transmettre au processus destinataire en espace utilisateur les tâches dans la liste d'attente. La fonction

7. `drivers/staging/android/binder`
8. Fonctions correspondantes implémentées dans `security/blare/binder_ops.c`

associée à ce *hook* propage les tags (*itag* et *xptag*) associés à la transaction vers le processus destinataire. Un contrôle de la légalité du flux est effectuée. Si le flux est illégal alors une alerte est levée. Afin que l'alerte soit plus compréhensible, le flux correspondant à l'alerte est décrite comme un flux de processus ayant émis la transaction vers le processus destinataire. De plus, une transaction est un conteneur volatile implémenté sous la forme d'une structure uniquement connue du noyau. Sa description dans une alerte n'apportera ainsi aucune information utile à un analyste. Selon les transactions, les informations liées à l'émetteur ne sont pas ajoutées dans la transaction. C'est le cas, par exemple des réponses retournées par un appel distant. Nous avons modifié le Binder afin que cette information soit toujours comprise dans la transaction.

Remarque sur le suivi de flux d'information dans le Binder

Une autre manière de propager les tags entre les processus aurait été de le faire directement à l'envoi de la transaction par le processus émetteur. Cependant, nous avons opté pour un suivi de flux en deux temps. Les tags sont propagés du processus émetteur vers la transaction puis de la transaction vers le processus destinataire. Chaque action effectuée par le noyau se fait toujours dans le contexte d'un processus. Pour les appels système, il s'agit du processus ayant effectué l'appel système. Lors de l'envoi de la transaction, le Binder s'exécute ainsi dans le contexte du processus émetteur. Propager directement les tags à l'envoi signifie modifier la structure associée au processus destinataire tout en étant dans le contexte de l'émetteur. Cette pratique n'est cependant pas conseillée dans la programmation noyau. Nous aurions également pu faire la propagation lors du traitement des tâches de la liste d'attente. Cela se fait dans le contexte du processus destinataire mais le risque est la possibilité de changement du contenu du processus émetteur entre l'envoi et la réception de la transaction. Cela pourrait par exemple permettre d'échanger des données tout en faisant croire à KBlare qu'aucune d'entre elles n'est sensible.

En optant pour la propagation en deux temps des tags, nous ne modifions pas la structure d'un processus autre que celui qui sert de contexte d'exécution et nous maintenons une cohérence entre les identifiants des informations partant du processus émetteur et ceux des informations récupérées par le destinataire.

Afin de prendre en compte les flux d'information s'opérant via le Binder, nous avons étendu le *framework* LSM. Nous avons ajouté deux nouveaux *hooks* permettant d'intercepter l'envoi et la transmission des transactions. Dans le code du Binder, les transactions représentent les appels de méthodes distantes ainsi que le retour de ces appels. Nous avons ensuite ajouté deux fonctions supplémentaires à KBlare qui correspondent à ces nouveaux *hooks* et effectuent la propagation des tags entre les applications qui communiquent via le Binder.

3.4.3 Exécution des applications Android

3.4.3.1 Interprétation de code vs exécution

Dans Blare, la prise en compte de l'exécution d'un code se fait en interceptant les appels système `execve` et `mmap` quand la page est marquée comme étant exécutable. Sous Android, à l'exception des applications nécessaires au fonctionnement système, les applications tournent au sein d'instance de la machine virtuelle dalvik. Pour comprendre comment ces instances sont créées, il est nécessaire de comprendre la séquence de démarrage d'Android. La figure 3.5 illustre cette séquence.

Comme pour les systèmes de type Linux, le premier processus lancé est `init` qui est en charge d'initialiser l'environnement Android. Les initialisations qu'il effectue sont définies dans les fichiers `init.*rc` à la racine du système de fichier. Il définit par exemple les variables d'environnement et monte les partitions. Il lance également les différents processus démons du système tels que `servicemanager`, `surfaceflinger` et `Zygote`. À son lancement, ce dernier lance une instance de la machine virtuelle dalvik, crée un *socket* serveur, charge un ensemble de ressources et classes Java, se duplique pour créer le processus `system_server` et attend les connexions sur le *socket* créé. `System_server` est le processus qui exécute les différents services du système (ex : Activity Manager). À sa création, il les lance donc et les enregistre auprès de `servicemanager` afin qu'ils puissent être appelés par les différentes applications du système.

Zygote est un processus clé pour l'exécution des différentes applications Android car c'est à partir de ce processus que les autres applications sont lancées. Par exemple, lorsque l'utilisateur lance une nouvelle application via le menu principal du téléphone, l'application `Lancher` appelle la fonction `startActivity` définie dans `Activity Manager`. À l'exécution de la méthode, ce dernier envoie une requête à `Zygote` pour lancer l'application. À la réception de la requête, ce dernier se duplique [9] et lance dans le processus fils la nouvelle application.

Au l'exécution d'une nouvelle application, le processus l'exécutant charge le code de l'application en mémoire. Ce code est sous format *dalvik* et n'est pas exécuté, comprendre via l'appel système `execve` ou `mmap` et marqué exécutable, mais chargé comme une simple donnée en mémoire. L'exécution des applications se traduit par la lecture de leur code en mémoire et son interprétation par la machine virtuelle dalvik. Comme aucun appel système synonyme d'exécution de code pour Blare n'est effectué alors Blare, dans sa version avant nos modifications, ne détectait jamais l'exécution des applications Android. Cela avait deux conséquences directes. Premièrement, l'exécution des applications Android dont le code est marqué comme sensible ne causait pas le marquage des processus comme exécutant du code sensible. L'*itag* des processus exécutant les applications Android ne contenait jamais d'identifiant négatif. Ensuite, il était impossible d'appliquer une politique par application car du point de vue de Blare ils exécutent tous le même programme (`app_process`).

9. appel système `fork`

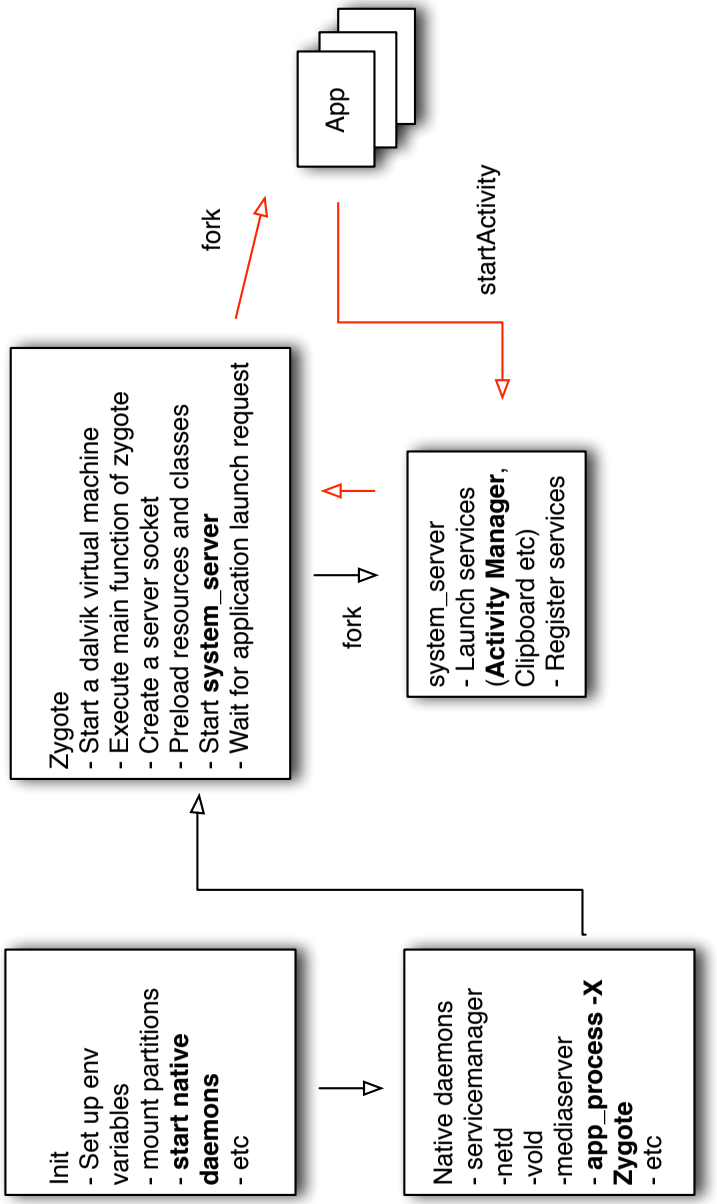


FIGURE 3.5 – Séquence de démarrage d'Android

3.4.3.2 Prise en compte de l'exécution des applications Android

Pour détecter l'exécution des applications, nous avons introduit un système de coopération entre la machine virtuelle Dalvik et KBlare. Il permet à la machine virtuelle de notifier KBlare quand elle va interpréter du code et donc permettre à KBlare de voir quand une application est exécutée. À la réception de la notification, KBlare met ainsi à jour les tags convenablement et applique la politique de flux associée à l'application.

L'exécution d'une application se traduit par le chargement des ressources de l'application en mémoire ainsi que de son code. Lorsqu'une application Android est installée, le système extrait à partir de son `apk` une version optimisée du code. C'est cette version optimisée qui est chargée en mémoire et interprétée par la machine virtuelle. Un bon point d'insertion pour la notification semble ainsi être le chargement de cette version optimisée. Aucune documentation technique n'existe sur le fonctionnement de Dalvik mais en analysant son code source, nous trouvons que le chargement est effectué par la fonction `dvmDexFileOpenFromFd` (listing 3.4). Cette fonction charge en mémoire le code optimisé d'une application et analyse syntaxiquement son contenu. Elle est appelée par deux autre fonctions [10] qui ouvrent un fichier `jar` ou `dex`, crée la version optimisée du code à partir du contenu du fichier ouvert et le charge en mémoire. Cela correspond bien à l'exécution d'une application. Nous avons ainsi ajouté la notification, fonction `odex_is_mapped`, au corps de la fonction `dvmDexFileOpenFromFd`.

La figure 3.6 illustre les différentes étapes de la prise en compte de l'exécution de l'application. Les tags vert et bleu correspondent respectivement au contenu du fichier `.dex` et à ce même contenu lorsqu'il est exécuté par un processus. Le tag marron correspond à la politique de sécurité de l'application. Le système de notification est similaire à une architecture client/serveur. Au démarrage du système, KBlare se met en écoute des notifications provenant des machines virtuelles Dalvik. (a) Lorsqu'un processus initie l'exécution d'une application (chargement du fichier `.dex` en mémoire), elle charge le contenu du fichier `.dex` en mémoire. KBlare voit ce chargement et marque le processus qui va exécuter l'application comme contenant le contenu du fichier .dex. (b) La machine virtuelle notifie ensuite KBlare qu'elle va interpréter le contenu du fichier `.dex`, c'est-à-dire l'exécuter. (c) et (d) À la réception de la notification, KBlare considère que le processus contenant la machine virtuelle ayant envoyée la notification va exécuter l'application dont le code est stocké dans le fichier `.dex`. KBlare met donc à jour les tags du processus. Il indique que le processus contient la version exécutée du contenu du fichier `.dex` et applique à ce processus la politique de flux associée au code que la machine virtuelle va interpréter.

Implémentation de la notification de l'exécution d'une application

La notification est envoyée grâce à un canal de communication entre la machine virtuelle Dalvik et KBlare. Ce canal est implémenté avec Generic Net-

10. `dvmJarFileOpen` et `dvmRawDexFileOpen` respectivement définies dans `dalvik/vm/JarFile.cpp` et `dalvik/vm/RawDexFile.cpp`

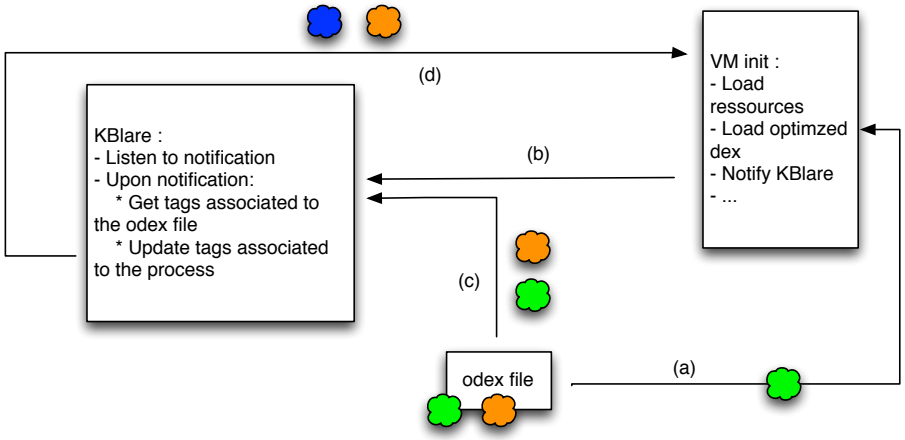


FIGURE 3.6 – Mécanisme de notification de l'exécution d'une application Android

link [52] qui lui utilise les sockets Netlink [63]. Netlink est un mécanisme de communication qui permet de faire communiquer entre eux des morceaux de code qui sont dans le noyau ou en espace utilisateur. La communication entre ces différentes entités se font dans des bus Netlink, appelés également protocoles. Il existe un nombre défini de bus dans les systèmes Linux et leur nombre maximum est limité à 32. Certains d'entre eux sont par exemple utilisés par des outils de gestion du trafic réseau tels que `iproute2` [53]. Generic Netlink est un mécanisme de communication basé sur Netlink et plus précisément un multiplexeur construit au dessus d'un bus Netlink [76]. Les canaux de communication sont appelés familles et il est possible d'en définir jusqu'à 65520 pour un seul et même bus Netlink. Nous présentons dans ce qui suit le système de coopération permettant de faire communiquer les machines virtuelles Dalvik et KBlare. Plus précisément, nous présentons la déclaration de la famille servant de canal de communication, le comportement associé à la réception d'une notification et la mise en attente.

Comme écrit précédemment, le système de notification est une architecture client/serveur. KBlare est le serveur qui se met en attente des notifications d'exécution d'application et les machines virtuelles Dalvik sont les clientes qui envoient les notifications au serveur. La création d'un serveur d'écoute avec Generic Netlink se fait en trois étapes : création d'une famille, définition des commandes reconnues par le serveur et enregistrement de la famille.

Afin que KBlare puisse recevoir les notifications, nous définissons dans son code une famille servant de canal de communication. Le listing 3.5 montre la déclaration de la famille. Nous déclarons en premier les attributs possibles d'un message. Nous en déclarons deux : `DOC_EXMPL_A_MSG` et `DOC_EXMPL_A_INT`. Ensuite, nous définissons le type associé aux attributs. Le premier attribut est une

```
int dvmDexFileOpenFromFd(int fd, DvmDex** ppDvmDex)
{
    DvmDex* pDvmDex;
    DexFile* pDexFile;
    MemMapping memMap;
    int parseFlags = kDexParseDefault;
    int result = -1;

    if (gDvm.verifyDexChecksum)
        parseFlags |= kDexParseVerifyChecksum;

    if (lseek(fd, 0, SEEK_SET) < 0) {
        ALOGE("lseek rewind failed");
        goto bail;
    }

    if (sysMapFileInShmemWritableReadOnly(fd, &memMap) != 0) {
        ALOGE("Unable to map file");
        goto bail;
    }

    // Nofify KBlare
    odex_is_mapped(fd);
    ...
}
```

Listing 3.4 – Fonction de chargement en mémoire du code d'une application

chaîne de caractère et le second un entier non signé sur 32 bits. Enfin, nous déclarons la famille : variable `doc_exmpl_gnl_family`. Chaque famille doit avoir un identifiant unique [11]. Le champ `id` contient l'identifiant numérique de la famille. En lui associant la valeur `GENL_ID_GENERATE`, nous laissons au système le choix de l'identifiant. Le champ `hdrsize` définit la taille de l'en-tête des messages transitant par cette famille. Nous n'avons besoin d'aucune en-tête spécifique. Nous le mettons ainsi à 0. Le champ `name` définit le nom de la famille. Chaque nom doit être unique et elle permet aux clients de récupérer l'identifiant de la chaine. Les champs restant définissent le numéro de version de la famille et le nombre d'attributs.

Une fois la famille définie, il faut ensuite déclarer les commandes qu'elle supporte ainsi que les fonctions qui leur sont associées. Les commandes définissent les opérations supportées par le serveur. Nous en définissons un, `DOC_EXMPL_EXEC_DEX`, dans le listing 3.6 qui correspond à la notification de l'exécution d'une application. À cette commande, nous associons la fonction `doc_exmpl_exec_dex`.

11. champ `id`

```
1  // USE GENERIC NETLINK
2  /* Family declaration */
3  /* attributes */
4  enum {
5      DOC_EXMPL_A_UNSPEC,
6      DOC_EXMPL_A_MSG,
7      DOC_EXMPL_A_INT,
8      __DOC_EXMPL_A_MAX,
9  };
10 #define DOC_EXMPL_A_MAX (__DOC_EXMPL_A_MAX - 1)
11
12 /* attribute policy */
13 static struct nla_policy doc_exmpl_genl_policy[DOC_EXMPL_A_MAX + 1] = {
14     [DOC_EXMPL_A_MSG] = { .type = NLA_NUL_STRING },
15     [DOC_EXMPL_A_INT] = { .type = NLA_U32 },
16 };
17
18 #define VERSION_NR 1
19 static struct genl_family doc_exmpl_gnl_family = {
20     .id = GENL_ID_GENERATE,
21     .hdrsize = 0,
22     .name = "BLARE_COOP", // family name
23     .version = VERSION_NR, //version number
24     .maxattr = DOC_EXMPL_A_MAX,
25 };
26 /* End of family declaration */
```

Listing 3.5 – Déclaration d'une famille Generic Netlink dans le noyau pour le mécanisme de coopération

Cette fonction récupère à partir de la charge utile du message reçu par le serveur un entier représentant le descripteur du fichier chargé par la machine virtuelle (ligne 24 à 26). À partir de ce descripteur, il récupère la structure correspondant au fichier puis met à jour les tags associés au processus (ligne 36 à 38).

Enfin, nous enregistrons la famille et les opérations qu'elle supporte (listing 3.7). Ces enregistrements sont effectués dans la fonction `netlink_init` qui est appelée pendant l'initialisation des modules de noyau (ligne 22).

Du côté client, la communication avec KBlare est plus simple. Elle se traduit par l'envoi d'un message avec la commande `DOC_EXMPL_EXEC_DEX` et le descripteur de fichier correspondant au code chargé en mémoire. La bibliothèque `libnl` [11] offre une API les sockets Netlink et Generic Netlink. Cependant, dans la version actuelle du code, nous avons utilisé les fonctions définies dans l'exemple du client fourni par A. Keller dans [64]. Le listing 3.8 présente une version épurée [12] du code servant à la notification de l'exécution des applications Android. La variable `req` est le message Netlink à envoyer dans le noyau.

12. sans vérification des codes d'erreur

Il contient l'en-tête du message Netlink, celui de Generic Netlink et la charge utile du message. Rappelons que Generic Netlink n'est qu'un multiplexeur au dessus d'un bus Netlink. L'envoi d'une notification se fait en plusieurs étapes.

1. Récupérer l'identifiant du canal de communication (lignes 13, 14). Il est nécessaire afin d'envoyer le message dans le bon canal de communication.
2. Initialiser les en-têtes Netlink et Generic Netlink. Dans le cas de Generic Netlink, il s'agit de définir la commande à envoyer, `DOC_EXMPL_EXEC_DEX`.
3. Composer le message. Cela consiste à définir les attributs à passer dans le message ainsi que la valeur qui leur est associée. Nous définissons ainsi comme attribut un entier et le descripteur du fichier `dex` comme valeur associée (ligne 23 à 28).
4. Envoyer le message. Generic Netlink est un multiplexeur au dessus d'un bus Netlink. Nous déclarons ainsi un socket Netlink et envoyons le message avec ce socket.

Lorsqu'une machine virtuelle va interpréter le contenu d'un fichier `.dex`, il appelle la fonction `odex_is_mapped` (listing 3.4) en lui donnant en paramètre le descripteur du fichier `.dex`. Cette fonction envoie un message, via une famille de communication Generic Netlink, au noyau et plus précisément à KBlare. Le message contient deux informations. La première est une commande qui permet au destinataire de choisir le traitement à effectuer à la réception du message. La deuxième information est le descripteur du fichier `.dex`. Il permettra à KBlare d'identifier le fichier dont le contenu va être interprété par la machine virtuelle et propager les tags de ce fichier vers le processus contenant la machine virtuelle. À la réception du message dans le noyau, ce dernier extrait du message la commande que le message contient. KBlare ayant associée à la commande, la fonction `doc_exmpl_dex` (listing 3.6), le noyau transfert le traitement du message à cette fonction. La fonction récupère à partir du message le descripteur de fichier et à partir de ce descripteur, il récupère la structure représentant le fichier contenant le code qui va être interprété. À partir de cette structure, KBlare récupère les tags associés aux fichiers et les propage comme dans le cas d'une exécution au processus ayant envoyé la notification. À la réception du message, KBlare tourne dans le contexte du processus ayant envoyé la notification. Ce sont donc les tags de ce processus qui sont mis à jour.

```
/* commands: enumeration of all commands (functions),
 * used by userspace application to identify command to be executed
 */
enum {
        DOC_EXMPL_C_UNSPEC,
        DOC_EXMPL_C_EXEC_DEX,
        __DOC_EXMPL_C_MAX,
};

#define DOC_EXMPL_C_MAX (__DOC_EXMPL_C_MAX - 1)

static int doc_exmpl_exec_dex(struct sk_buff *skb_2, struct genl_info *info) {
        struct nlattr *na;
        struct sk_buff *skb;
        int rc;
        void *msg_head;
        int * mydata;
        struct file * dexfile;
        struct blare_file_struct *fstruct;

        if (info == NULL)
                goto out;

        na = info->attrs[DOC_EXMPL_A_INT];
        if (na) {
                mydata = (int *) nla_data(na);
                if (!mydata)
                        printk(KERN_INFO "[BLARE_NETLINK] error while"
                    "receiving data\n");
        } else {
                printk(KERN_INFO "[BLARE_NETLINK] no fd from userspace");
                goto out;
        }

        rcu_read_lock();
        dexfile = fcheck(*mydata);
        if (dexfile) {
                interp_exec(dexfile);
        } else {
                printk(KERN_INFO "[BLARE_NETLINK] could not find"
                "struct file of %d\n", *mydata);
        }
        rcu_read_unlock();
out:
        return 0;
}

/* Map command and function */
struct genl_ops doc_exmpl_gnl_ops_exec_dex = {
        .cmd = DOC_EXMPL_C_EXEC_DEX,
        .flags = 0,
        .policy = doc_exmpl_genl_policy,
        .doit = doc_exmpl_exec_dex,
        .dumpit = NULL,
};
```

Listing 3.6 – Définition d'une commande Generic Netlink pour notifier l'exécution d'une application

```
1  static int __init netlink_init() {
2      int rc;
3      printk(KERN_INFO "Blare init generic netlink");
4
5      /* register new family */
6      rc = genl_register_family(&doc_exmpl_gnl_family);
7      if (rc != 0)
8          goto failure;
9
10     /* register operations */
11     genl_register_ops(&doc_exmpl_gnl_family,
12             &doc_exmpl_gnl_ops_exec_dex);
13
14     return 0;
15
16 failure:
17     printk(KERN_INFO "[BLARE_NETLINK] an error occured while"
18             "inserting the generic netlink example module\n");
19     return -1;
20 }
21
22 __initcall(netlink_init);
```

Listing 3.7 – Enregistrement de la famille servant à la notification d'exécution des applications

```
int odex_is_mapped(int fd) {
    int nl_sd; // Netlink socket
    int id;
    int r;
    struct {
        struct nlmsghdr n;
        struct genlmsghdr g;
        char buf[256];
    } req;
    struct nlattr *na;

    /* Get the identifier of the family */
    nl_sd = create_nl_socket(NETLINK_GENERIC,0);
    id = get_family_id(nl_sd);

    /* Init netlink header */
    ...

    /* Init generic netlink header */
    req.g.cmd = DOC_EXMPL_EXEC_DEX;

    /* Compose message */
    na = (struct nlattr *) GENLMSG_DATA(&req);
    na->nla_type = DOC_EXMPL_A_INT;
    int mlength = sizeof(int);
    na->nla_len = mlength + NLA_HDRLEN;
    memcpy(NLA_DATA(na), &fd, mlength);
    req.n.nlmsg_len += NLMSG_ALIGN(na->nla_len);

    /* Send message */
    struct sockaddr_nl nladdr;
    memset(&nladdr, 0, sizeof(nladdr));
    nladdr.nl_family = AF_NETLINK;
    r = sendto(nl_sd, (char *)&req, req.n.nlmsg_len, 0,
            (struct sockaddr *) &nladdr, sizeof(nladdr));
    close(nl_sd);
    return 0;
}
```

Listing 3.8 – Notification de l'exécution d'une application par la machine virtuelle Dalvik

3.4.4 Description des flux observés

KBlare décrit les flux impliquant des données sensibles dans un format assez verbeux et sans fournir assez d'information parfois. La figure 3.7 est un exemple de description de flux émis par KBlare lorsqu'il détecte un flux d'un fichier vers un processus. Parmi les informations que nous aurions aimé avoir étaient le chemin complet du fichier, son identifiant et les informations liées au processus. Avoir le chemin complet du fichier permet à un analyste d'identifier avec précision le fichier impliqué dans le flux. L'identifiant du fichier, son `inode`, peut également s'avérer utile si le nom du fichier est amené à changer au cours de l'exécution du système.

Sous Android, nous avons ainsi mis les alertes sous la forme décrite en figure 3.8. Un message est découpé en trois parties séparées par le caractère `'>'`. Les deux premières parties décrivent le conteneur source et destination du flux. `C_TYPE` représente le type du conteneur (fichier, socket et processus). `C_NAME` est le nom du conteneur. Dans le cas des sockets, il s'agit de l'adresse IP associée à la socket. Dans le cas des processus, nous donnons à la fois le nom du `thread` courant et du processus. Les deux noms peuvent différer. Par exemple, les services dans `system_server` tournent dans des threads avec des noms différents. Avoir le nom des deux peut ainsi aider à comprendre quelle partie de l'application est à l'origine du flux.

3.4.5 Outils en espace utilisateur

Il existe sous Linux des outils en ligne de commande pour manipuler les tags Blare associés aux fichiers. Leur code est disponible dans la branche `master` du dépôt `kblare-tools` [38]. Ces outils ont été portés sous Android, branche `android_version` du même dépôt, et une version sous forme d'extension d'un navigateur de fichier Android a également été développée lors d'un stage de 2 mois par Q. Dion. L'extension permet de manipuler via l'interface tactile d'un appareil Android les tags associés aux fichiers et évite ainsi d'avoir à ouvrir une interface en ligne de commande. Dans le cadre d'un autre stage, T. Saliou a développé Blare Policy Manager (BPM). BPM implémente les fonctions de composition et de vérification des politiques écrites en BSPL et applique la politique résultant de la composition sur le système. Nous avons utilisé BPM dans [20] pour composer et appliquer les politiques d'application tierce sur un téléphone avec l'environnement AndroBlare.

```
Process with pid 98 running cat made an illegal READ access to
file toto.txt
```

FIGURE 3.7 – Exemple de message levé par Blare sous Linux

```
C_TYPE C_NAME C_ID > C_TYPE C_NAME C_ID > ITAG
```

FIGURE 3.8 – Format des flux observés par Blare sous Android

3.5 AndroBlare : environnement d'analyse d'applications Android

Dans cette thèse, nous avons utilisé AndroBlare en tant que environnement d'analyse d'application Android. La figure 3.9 illustre l'environnement ainsi que l'usage que nous en aurons dans les chapitres suivants. Pour analyser les applications, nous utilisons un téléphone Nexus S faisant tourner la version AOSP d'Android Ice Cream Sandwich. Nous avons ajouté au téléphone tout l'environnement AndroBlare : le module de sécurité dans le noyau, le système de notification d'exécution d'application Android et les outils utilisateurs pour manipuler les tags. Nous avons également rajouté les application Super User afin de recevoir une notification visuelle à chaque fois qu'une application demande les accès `root` ainsi que `busybox`, un équivalent plus complet de `toolbox`.

Dans cet environnement, nous exécutons toute application dont nous souhaitons analyser le comportement. Ce comportement est la propagation des données de l'application dans le système et est obtenu grâce au suivi de flux d'information effectué par AndroBlare. Pour chaque application à analyser, nous répétons le processus suivant. Nous installons l'application sur le téléphone puis marquons son fichier `apk` avec un identifiant unique. Nous considérons que le fichier `apk` est la source des données appartenant à l'application car il contient à la fois son code et les ressources qu'elle utilise telles que les images. Nous exécutons ensuite l'application et observons avec AndroBlare comment les données de l'application se propagent dans le système. À partir des flux observés, nous pouvons effectuer deux types d'action : construire une représentation compacte et humainement compréhensible des flux observés (chapitre 4) ou détecter l'exécution d'instance de malware (chapitre 5).

Nous laissons les tags des autres conteneurs d'information à leur valeur par défaut, ensemble vide, sauf pour les fichiers `servicemanager` et `surfaceflinger` dans `/system/bin`. À l'exception de ces deux fichiers, les autre conteneurs ne contiennent donc aucune information sensible du point de vue d'AndroBlare et n'ont aucune politique de flux d'information. Quant aux deux fichiers, nous leur assignons l'*itag* {0} pour signifier que les processus les exécutant sont à ignorer pendant le suivi de flux d'information. L'identifiant 0 n'est pas utilité dans le modèle de Blare pour identifier une information à surveiller. Dans l'implémentation, nous nous servons de cette valeur pour indiquer à Blare quand ignorer des flux d'information. Le but est d'éviter une trop grande surapproximation des flux observés par AndroBlare. Les applications `servicemanager` et `surfaceflinger` sont deux processus clés du système Android. Le premier est un service d'annuaire répertoriant tous les autres services du système. Les applications souhaitant utiliser ces services doivent demander au `servicemanager` leur référence avant de pouvoir les utiliser. Le second est en charge de dessiner sur

l'écran toute interface graphique. Les applications sont uniquement conscients de ce qu'ils veulent afficher à l'écran. Elles le transmettent à `surfaceflinger` qui lui composera les affichages des différentes applications et affichera le rendu final sur l'écran de l'appareil.

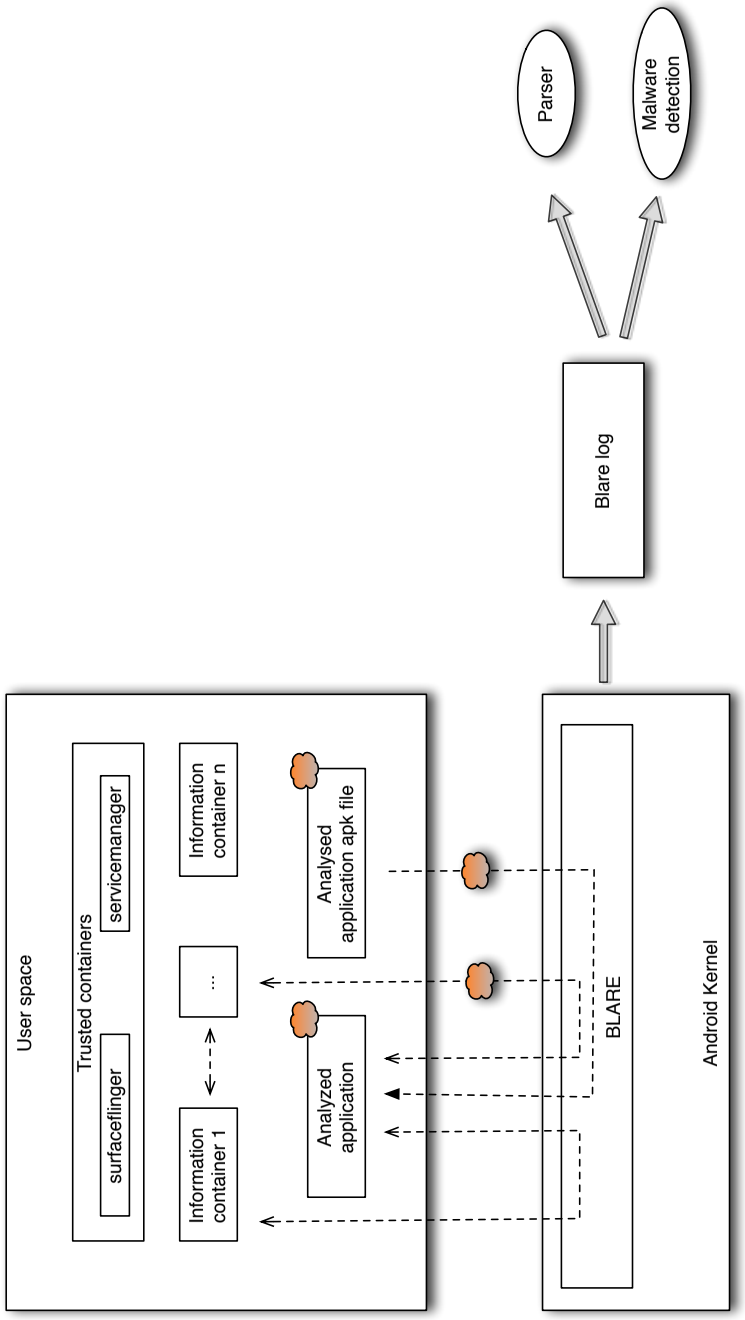


FIGURE 3.9 – AndroBlare : environnement d'analyse d'application Android

Résumé

Dans ce chapitre, nous avons présenté Blare un système de détection d'intrusion paramétré par une politique de flux d'information, le modèle sur lequel il est basé et le processus de portage de Blare sous Android (AndroBlare). AndroBlare utilise le même modèle théorique que Blare mais diffère de ce dernier au niveau de l'implémentation. Lors du portage de Blare sous Android, nous avons ajouté la prise en compte de deux types de flux d'information : flux d'information via le Binder et l'interprétation du code des applications Android par la machine virtuelle Dalvik. Les applications Android n'utilisent pas les mécanismes de communication fournis par le noyau Linux pour communiquer entre elles. À la place, elles utilisent différents mécanismes fournis par le framework Java qui reposent tous sur le Binder. Afin de suivre les flux d'information entre les applications, nous avons pris en compte ce mécanisme de communication dans AndroBlare. Sa prise en compte a consisté à ajouter deux nouveaux *hooks* à LSM et à définir dans le code de KBlare les fonctions correspondant à ces *hooks*. Le deuxième type de flux d'information pris en compte dans AndroBlare concerne l'exécution des applications et plus précisément l'interprétation de leur code par la machine virtuelle Dalvik. Les applications Android ne sont pas des applications natives et leur exécution était donc invisible à Blare. Lorsqu'une application est `exécutée`, son code est en réalité interprétée par la machine virtuelle Dalvik. Le problème posé par cette interprétation du code est que l'exécution était invisible et donc KBlare ne pouvait mettre à jour correctement les tags des processus `exécutant` du code sensible. Aucun processus n'était marqué comme exécutant du code et aucune politique ne pouvait être appliquée par application. Afin de permette à KBlare d'observer l'exécution d'une application Android, nous avons ainsi ajouté un mécanisme de coopération entre la machine virtuelle Dalvik et KBlare. Cette coopération consiste à notifier KBlare via les sockets Netlink de l'*exécution* d'une application par la machine virtuelle.

Lors de la présentation du modèle de Blare, nous avons également expliqué comment nous avons défini manuellement une politique de flux d'information pour le système Android. La politique identifie 150 informations à protéger et 186 conteneurs d'information pouvant accéder à ces informations ou les stocker. Cette politique a fait l'objet d'une publication à ICC 2012 [16].

Chapitre 4

Graphes de flux système

Pour chaque flux d'information observé qui implique une donnée sensible, Blare ajoute une entrée décrivant le flux dans un journal. Au fur et à mesure que le système s'exécute, le taille du journal augmente et peut devenir rapidement difficile à analyser pour un être humain. Quelques minutes d'exécution du système peut suffire pour avoir des milliers d'entrées dans le journal de Blare. Nous proposons donc dans ce chapitre une structure appelée System Flow Graph pour représenter les flux observés de manière plus compacte et plus compréhensible.

4.1 Graphe de flux système

Un graphe de flux système, que nous abrégeons SFG (*System Flow Graph*) dans le reste du document, est un multigraphe orienté $G = (V, E)$ où les nœuds représentent des conteneurs d'information et les arcs des flux d'information entre les conteneurs.

Un nœud $v \in V$ a trois attributs $v.type$, $v.name$ et $v.id$ qui représentent respectivement le type du conteneur (fichier, processus et socket) correspondant au nœud, son nom et son identifiant du conteneur dans le système. Selon le type du conteneur, le nom du nœud est soit le chemin complet du fichier, le nom du processus concaténé à celui du thread ou l'adresse IP associée à la socket. L'identifiant est le numéro d'i-node pour les fichiers et le PID pour les processus.

Un arc $e \in E$ a deux attributs $e.timestamp$ et $e.flow$ qui correspondent respectivement à la liste des moments auxquels le flux a été observé et les identifiants des informations impliquées dans le flux. Un arc représente un flux d'information unique du conteneur représenté par sa source vers le conteneur représenté par sa destination. Le flux peut être observé plusieurs fois et inclut à chaque observation les mêmes informations sensibles, i.e les identifiants des informations qui se propagent sont les mêmes à chaque observation.

Un SFG est un multigraphe car il peut contenir des arcs parallèles. Des arcs parallèles sont des arcs reliant les même nœuds. Un SFG a deux arcs parallèles

```
[1] file blob 18 > process cat 19 > {1, 2}
[2] process woman 20 > file blob 18 {3}
[3] process woman 20 > file blob 18 {3}
[4] file blob 18 > process cat 19 > {1, 2, 3}
```

FIGURE 4.1 – Exemple de flux d'information causant l'apparition d'arcs parallèles dans les SFG

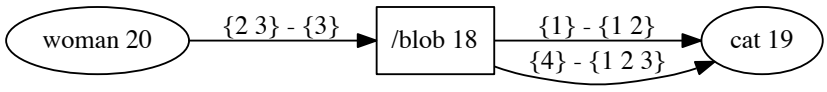


FIGURE 4.2 – Exemple de SFG avec des arcs parallèles

lorsque Blare a observé deux flux d'information ayant la même source et la même destination mais n'impliquant pas les mêmes informations sensibles. La figure 4.1 décrit des flux d'information causant la création d'arcs parallèles dans les SFG. La première et dernière entrées décrivent des flux ayant la même source et la même destination : de `blob` à `cat`. Cependant, le contenu de `blob` a été modifié par le deuxième et troisième flux observés par Blare et ainsi le dernier flux implique une information en plus par rapport à la première. Ces deux flux ne peuvent donc être considérés comme un flux unique et sont représentés avec deux arcs. Le deuxième et le troisième flux impliquent les mêmes informations sensibles et représentent donc un flux d'information unique. Ils sont représentés par un unique arc.

Graphe de flux système vs. graphe de dépendance

Dans [66], Samuel T. King et Peter M. Chen utilisent un graphe de dépendance pour analyser les intrusions dans un système. Un graphe de dépendance décrit sous la forme d'un graphe les flux d'information entre les objets du système. Les nœuds sont les objets du système et un arc entre deux nœuds signifie qu'il y a eu un flux d'information entre les objets représentés par les nœuds source et destination de l'arc. Lorsqu'une intrusion a été détectée sur un objet du système, ils construisent le graphe de dépendance et l'analysent afin d'avoir un début de diagnostic de l'intrusion. Le graphe de dépendance est construit à partir des flux d'information qui ont eu lieu jusqu'à la détection de l'intrusion et sa construction est similaire à celle d'un SFG.

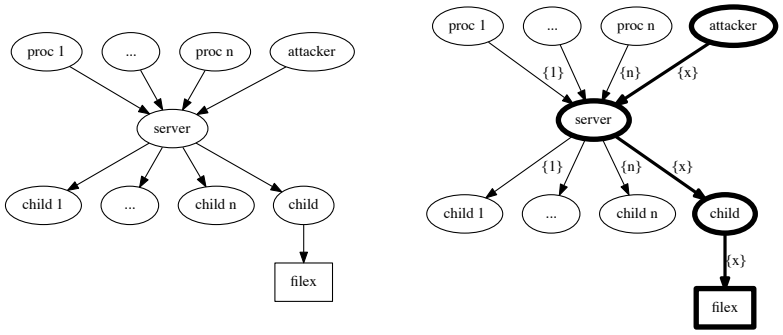
Si un SFG et un graphe de dépendance décrivent tous les deux des flux d'information entre les objets du système, le SFG apporte cependant plus de

précision car il est plus centré sur les informations que ne l'est un graphe de dépendance. Un graphe de dépendance prend en compte tout flux d'information s'opérant dans le système et ne fait aucune distinction entre les informations qui se propagent. Par contre, un SFG lui ne prendra en compte que les flux d'information impliquant les données à surveiller dans le système ce qui permet de mieux filtrer les flux d'information à analyser lors d'une intrusion et raccourcir la durée de l'analyse. Nous démontrons ce gain en représentant la suite d'évènement suivante avec un SFG et un graphe de dépendance pour analyser une intrusion.

Soient un processus *server* exécutant une application traitant des données à la demande d'applications clientes et $filex$ un fichier dont l'accès est limité à *server* et dont l'intégrité doit être préservée. Sur le même système tournent n processus bénins exécutant différents clients de *server*. À chaque requête cliente que *server* reçoit, *server* se duplique et attribue le traitement de la requête au processus fils résultant de la duplication. Un autre processus *attacker* exécute également une application cliente de *server* sur le système mais contrairement aux autres clients, il s'agit d'un processus malveillant dont le but de corrompre le contenu de $filex$. Pour corrompre le contenu de $filex$, il envoie une requête malicieuse à *server* qui lors de son traitement va forcer le processus fils de *server* à corrompre $filex$. Nous supposons que la détection ait lieu au moment de la corruption et que l'attaque soit menée seulement après que les n processus aient chacun envoyé au moins une requête au serveur. Cette dernière supposition permet d'avoir le pire des cas dans lors de l'analyse car tous les clients sont d'éventuels suspects.

En supposant que ces évènements se soient passés dans l'environnement d'analyse utilisée dans [66], nous construisons le graphe de dépendance illustré dans la figure 4.3.a. En analysant ce graphe, le seul diagnostic que nous déduisons est que l'un des processus clients a pu influencer le serveur à modifier le contenu de $filex$. Tous les processus clients sont de potentiels suspects car ils ont tous envoyés des données au serveur avant que l'attaque ne soit détectée.

Si nous supposons cette fois que les même évènements se soient produits dans un environnement (Andro)Blare et que pour chaque client un identifiant unique est associé aux données qu'il envoie alors nous aurions un SFG illustré dans la figure 4.3.b. Nous supposons que l'identifiant x est associé aux données provenant de l'application malveillante. En analysant le SFG, nous déduisons que $filex$ a été contaminé avec des données identifiées par x. En filtrant le SFG pour ne garder que les arcs impliquant les données identifiées par x, nous obtenons la partie en gras du SFG qui indique clairement que le processus *attacker* est à l'origine de l'attaque.



(a) (b)

FIGURE 4.3 – Graphe de dépendance et SFG représentant les flux d'information ayant précédé la corruption d'un fichier $filex$

4.2 Quelques opérations utiles sur les SFG

Dans les chapitres suivants, nous effectuerons certaines opérations sur les SFG. Nous présentons dans cette section ces opérations qui nous serviront par la suite.

4.2.1 Intersection de deux SFG : $g_1 \sqcap g_2$

L'intersection de deux SFG est le SFG dont les arcs sont l'ensemble des arcs en commun aux deux SFG. La notion d'arcs en commun de deux SFG désigne le fait qu'une partie des arcs des deux SFG sont les mêmes. Nous considérons qu'un arc $e_1 \in g_1$ est le même qu'un arc $e_2 \in g_2$ si :

- les deux arcs impliquent les même informations dans le flux décrit par chacun des arcs ;
- leurs nœuds de départ respectifs représentent le même conteneur d'information ;
- leurs nœuds d'arrivée respectifs représentent le même conteneur d'information.

Deux nœuds représentant le même conteneur d'information doivent avoir le même type et le même nom. Nous ne prenons pas en compte l'identifiant système des conteneurs d'information car ils sont spécifiques à chaque système ou à une exécution. Un processus n'existe tout au plus que le temps d'exécution du système. Son identifiant est donc lié à cette exécution. L'identifiant d'un fichier est lié au système où il se trouve. Il est généré automatiquement à la création du fichier. Un même fichier sur deux systèmes différents a donc deux identifiants différents.

Dans le pire des cas, le coût de cette opération est de l'ordre de $O(n \times m)$, n et m étant le nombre d'arc dans g_1 et g_2, car nous comparons deux à deux les

arcs des deux SFG.

4.2.2 Inclusion d'un SFG dans un autre : $g_1 \sqsubseteq g_2$

Soient deux SFG g_1 et g_2. Nous considérons que g_1 est inclus dans g_2 si tous les arcs de g_1 sont des arcs en commun avec g_2. Tout comme l'intersection, le coût de cette opération est dans le pire des cas de l'ordre de $O(n \times m)$, n et m étant le nombre d'arcs des deux SFG.

4.2.3 Nœuds et arcs d'un SFG

Nous définissions également les fonctions `node` et `edges` comme étant les fonctions retournant respectivement l'ensemble des nœuds et des arcs d'un SFG.

4.3 Construction d'un graphe de flux système

Pour obtenir la représentation compacte du journal de Blare, nous avons développé un outil qui prend en entrée un journal de Blare et donne en sortie la représentation sous forme de SFG. L'algorithme 1 décrit la transformation du journal en SFG. Nous définissons la fonction `to_edge` comme étant une fonction qui calcule l'arc correspondant à une entrée d'un journal de Blare et utilisons l'opération $\triangleq$ pour exprimer le fait que deux arcs représentent le même flux d'information unique. Deux arcs représentent le même flux d'information unique si leurs nœuds sources représentent le même conteneur d'information ainsi que leurs nœud de destination. Durant la construction d'un SFG, nous considérons que deux nœuds $v1$ et $v2$ représentent le même conteneur d'information si l'un des cas suivants est vrai :

- $v1$ et $v2$ sont des fichiers, $v1.id$ est égal à $v2.id$ et les fichiers correspondant sont sur la même partition ;
- $v1$ et $v2$ sont des processus et $v1.id$ est égal à $v2.id$;
- $v1$ et $v2$ sont des sockets qui sont liées à la même adresse IP.

Nous utilisons l'identifiant au lieu des noms dans le cas des processus et des fichiers car les noms peuvent changer durant l'exécution du système. Les identifiants identifient de manière unique chaque objet et ne changent pas durant l'exécution du système. Utiliser le nom aurait ainsi créé une confusion pour l'analyste car il ne pourrait plus retracer correctement la propagation des flux dans le système. Par exemple, durant l'installation d'une application, un fichier avec un nom aléatoire est créé dans `/data/dalvik-cache`. Ce fichier contient la version optimisée du code de l'application à installer et est renommé en se basant sur le nom de l'application à la fin de l'installation. Si le conteneur est impliqué dans un flux d'information avant et après son renommage, nous aurions un seul et même nœud pour le représenter avec les identifiants alors que nous aurions deux nœuds différents en utilisant les noms.

Pour chaque entrée du journal de Blare, nous vérifions si il existe un arc e_1 correspondant au flux décrit par l'entrée. Si c'est le cas, alors nous ajoutons le

timestamp de l'entrée courante à la liste des timestamps de $e1$. Si ce n'est pas le cas, nous ajoutons l'arc correspondant à l'entrée courante au SFG. L'ajout d'un nouvel arc implique également l'ajout de ses nœuds s'ils n'existaient pas dans le SFG ; c'est-à-dire qu'aucun des nœuds du SFG n'était égal aux nœuds source et destination du nouvel arc. La construction s'arrête quand il n'y a plus d'entrée à traiter et l'algorithme retourne le SFG.

Afin d'avoir une estimation du temps nécessaire pour construire un SFG à partir d'un journal d'AndroBlare, nous avons mesuré le temps d'exécution de notre algorithme sur 46 journaux différents. Ces journaux contiennent en moyenne 126000 entrées. Les mesures effectuées ont montré qu'il a fallu 25 minutes pour construire les 46 SFGs soit 1 minute et 50 secondes par SFG et 1145 entrées traitées à la seconde.

Algorithme 1 : Construction d'un SFG à partir des entrées d'un journal de Blare

Input : Journal de Blare
Output : SFG correspondant aux flux décrits dans le journal donné en entrée
begin
  $g \leftarrow$ empty SFG;
  forall the *entry ent in Blare log* **do**
    $e_0 \leftarrow$ `to_edge`(ent);
    $found \leftarrow False$;
    forall the $e_1 \in g$ **do**
      if $e_0 \triangleq e_1$ **then**
        $e_1.timestamp \leftarrow e_1.timestamp \cup e_0.timestamp$;
        $found \leftarrow True$;
        break;
    if $\neg found$ **then**
      Add e_0 to g;
  return g;

Compacité du SFG

Nous avons avancé au début de ce chapitre que le premier atout du SFG était de représenter de manière plus compacte les flux d'information observés par Blare. Cette compacité a pour conséquence de faciliter l'analyse des flux observés afin de comprendre ce qui se passe dans le système. Durant la thèse nous avons analysé plus d'une centaine d'application dont certaines étaient malicieuses et d'autres non. L'analyse de chacune d'entre elles a duré entre 2 à 5 minutes selon les applications et le journal de Blare obtenu à la fin de l'analyse contenait quelques milliers d'entrée, c'est-à-dire que Blare a observé des milliers de flux d'information et chacun des flux observés par une entrée dans le journal. À titre

d'exemple, l'analyse de 65 applications provenant de Google Play a produit en moyenne plus de 130000 entrées dans le journal de Blare alors que les SFG produits à partir de chaque journal ont en moyenne une centaine d'arcs. La raison de ce gain en compacité est que certains flux sont observés plusieurs fois durant l'exécution du système. Les échanges entre une application et le processus `system_server` sont par exemple répétés plusieurs fois car ce processus héberge les applications services du système fournies aux autres applications. Un autre exemple de ces flux répétés est la lecture ou l'écriture de données volumineuses dans un fichier.

4.4 Graphe de flux système : profil comportemental d'une application

Un SFG représente de manière compacte les flux d'information observés par (Andro)Blare. Dans cette thèse, nous proposons d'utiliser le SFG en tant que profil d'une application. Plus précisément, nous proposons d'utiliser le SFG pour décrire comment une information provenant d'une application se propage dans le système entier. Afin de construire le SFG, nous analysons avec AndroBlare comment l'information provenant d'une application sous surveillance se propage dans le système. Nous installons l'application, assignons un nouvel identifiant i à ses données, marquons son `apk` avec un *itag* égal à $\{i\}$, l'exécutons et analysons avec Blare comment ses données se propagent dans le système. Nous assignons un nouvel identifiant aux données de l'application afin d'identifier les flux d'information impliquant les données de l'application. L'`apk` contient toutes les ressources d'une application. Nous le considérons donc comme l'origine des données d'une application et le marquons avec l'*itag* $\{i\}$. À chaque fois que Blare observe un flux d'information impliquant une donnée identifiée par i il rajoute une entrée décrivant le flux observé. En utilisant l'algorithme 1, nous construisons ensuite le SFG correspondant aux flux observés. Le SFG résultant peut ensuite être analysé afin de comprendre les actions d'une application. Dans ce qui suit, nous proposons de construire le SFG d'un échantillon de malware Android et de l'analyser.

4.4.1 Analyse de DroidKungFu1 avec AndroBlare

DroidKungFu1 [61] est un malware Android découvert en 2011 sur les plateformes de téléchargement alternatives à Google Play. Il mène deux types d'attaque : le vol des données liées au téléphone (IMEI, numéro du téléphone et version du système d'exploitation) et l'ajout d'application sur le téléphone. Le vol des données est une attaque basique car l'application demande à l'installation les permissions requises pour accéder aux données et communiquer sur le réseau. Une fois installée, l'application collecte les données sensibles et les envoie dans une requête HTTP vers un serveur distant. En revanche, l'ajout d'application est plus complexe et les permissions demandées ne laissent en rien présager l'installation d'une nouvelle application. Pour installer d'autres

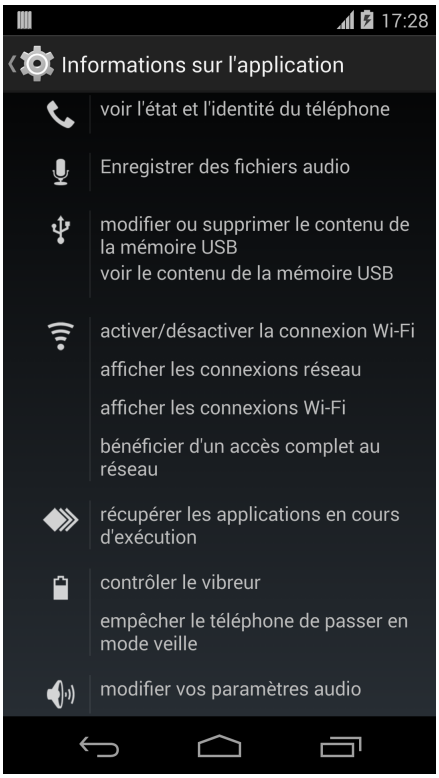


FIGURE 4.4 – Permissions demandées par un échantillons de DroidKungFu

applications, le malware élève ses privilèges en exploitant une vulnérabilité du système Android [31] ou en utilisant le binaire `su`[1].

L'échantillon que nous utilisons[2] provient de la collection Contagio [82] et a été proposé comme un client VoIP sur des plateformes alternatives à Google Play. La figure 4.4 est une capture d'écran des permissions demandées à l'utilisateur par l'échantillon. Parmi les accès demandés, nous remarquons l'accès à l'identité du téléphone (ex : IMEI et numéro de téléphone) et l'accès au réseau. Nous analysons l'échantillon avec AndroBlare comme décrit en section 3.5. Nous l'installons sur un téléphone, marquons son `apk`, l'exécutons et analysons avec AndroBlare comment ses informations se propagent dans le système.

Le code malveillant dans l'application est automatiquement exécuté dès que nous lançons l'application. Super User notifie que l'application a obtenu les droits `root` et au bout de quelques secondes, nous remarquons une nouvelle application, `Google SSearch`, dans le menu principal du téléphone (figure 4.5). Nous arrêtons l'analyse et créons à partir du journal de Blare le SFG de l'application. Le nombre d'entrées créées dans le journal est de 3563 et à partir de ces

1. switch user
2. Empreinte MD5 : 39d140511c18ebf7384a36113d48463d

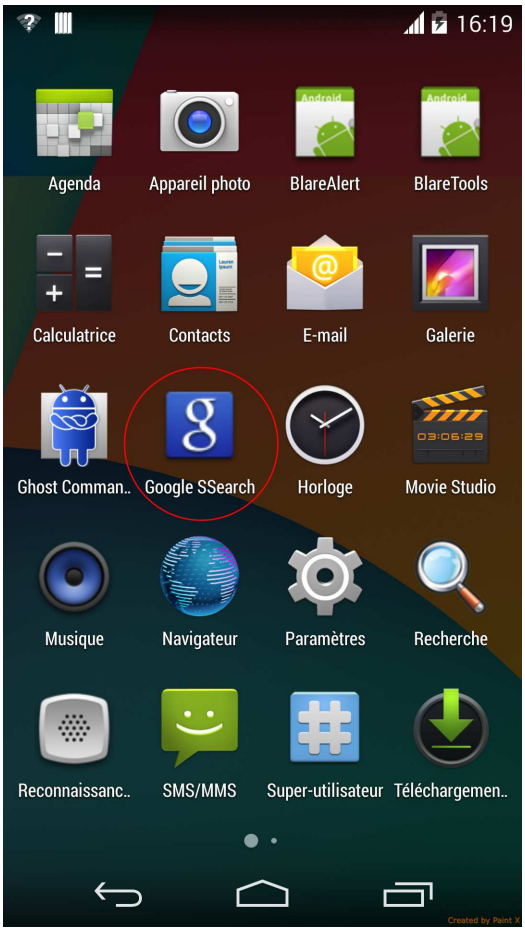


FIGURE 4.5 – Liste des applications dans le menu du téléphone après l'installation d'une nouvelle application par un échantillon de DroidKungFu

entrées nous construisons le SFG que nous analysons dans la section suivante.

4.4.2 Analyse du SFG de DroidKungFu1

La figure 4.6 illustre un extrait du SFG de l'échantillon 39d140511c18e-bf7384a36113d48463d. Le SFG entier est plus grand (106 arcs et 76 nœuds) mais la figure montre la partie la plus importante du point de vue de l'attaque (partie en gras). Le SFG a deux types de nœuds. Les ellipses représentent des processus tandis que les boîtes représentent des fichiers. Les arcs représentent toujours des labels mais pour des raisons esthétiques, nous nous sommes limités à afficher le nombre de fois que les flux correspondants aux arcs ont été observés ainsi que le timestamp de la première observation. L'application que nous avons

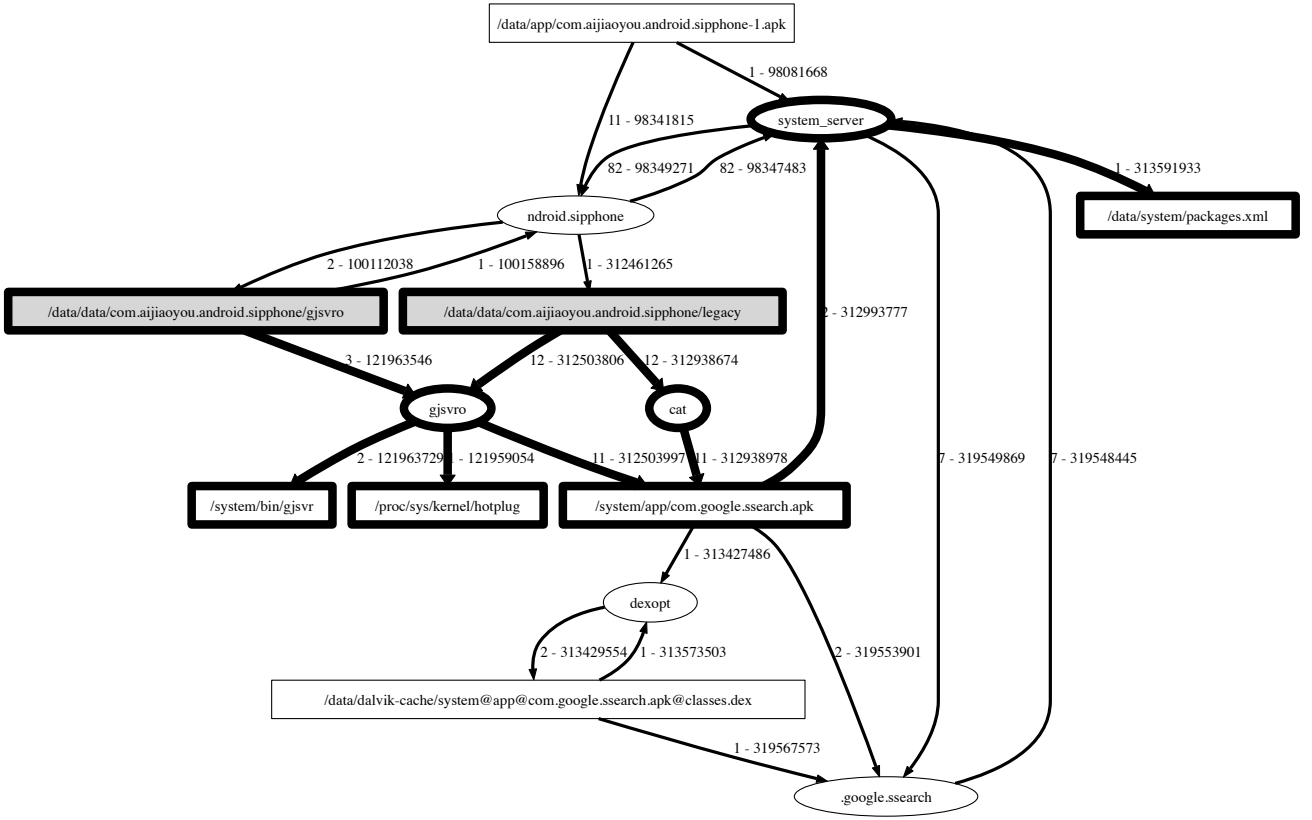


FIGURE 4.6 – Extrait du SFG d'un échantillon de DroidKungFu

analysé est le processus `ndroid.sipphone`. En analysant le SFG, nous pouvons déduire que deux applications ont été installées sur le téléphone.

Lorsque l'échantillon s'exécute, il crée deux fichiers `/data/data/com.aijaoyou.sipphone/gjsvro` et `/data/data/com.aijaoyou.sipphone/legacy`. À partir du contenu de ces fichiers, deux processus, `gjsvro` et `cat`, créent deux nouveaux fichiers dans la partition `system` : `/system/bin/gjsvro` et `/system/app/com.google.search.apk`. Cela indique l'installation d'une application native, `gjsvro`, et d'une application Android, `com.google.search.apk`. Ces deux fichiers n'existent pas par défaut sous Android, ce qui laisse supposer qu'ils ont été créés par l'échatillon du malware que nous analysons. Les flux qui suivent renforcent cette hypothèse car le contenu du fichier `apk` se propage vers le processus `system_server` qui lui le propage dans le fichier `packages.xml`. Le processus `system_server` exécute divers services du système dont celui en charge de l'installation des nouvelles applications, `Package Manager`. `Package Manager` observe la création de nouveaux fichiers dans le répertoire `/system/app` qui stocke les applications système. Si un fichier est créé dans ce répertoire, il lance l'installation du fichier. Le fichier `packages.xml` contient la liste des applications installées sur le téléphone. De plus, le contenu du fichier `apk` est également lu par le processus `dexopt` qui est en charge d'extraire la version optimisée du code d'une application à partir de son `apk`. Un nouveau processus, `google.search` accède ensuite à cette version optimisée ainsi qu'à l'`apk` créé par le malware ce qui indique l'exécution d'une nouvelle application.

Pour confirmer l'installation, nous analysons le fichier `packages.xml`. En calculant la différence de son contenu avant et après l'analyse de l'échantillon, nous remarquons une entrée décrivant une nouvelle application `com.google.ssearch` (listing 4.1). L'entrée indique que le code de l'application correspond au fichier `com.google.ssearch.apk` dans la partition `system`. Elle indique aussi l'UID associé à l'application : 10059. À l'installation d'une application, le système lui associe un nouvel UID dont la valeur est le dernier UID associé à une application incrémenté de 1. L'UID associé à l'échantillon que nous avons analysé est 10058. Cela signifie donc que l'application `com.google.ssearch` a été installée après l'échantillon que nous avons analysé.

```
<package name="com.google.ssearch" codePath="/system/app/com.google.ssearch.apk"
    nativeLibraryPath="/data/app-lib/com.google.ssearch" flags="48709"
    ft="147594a8de0" it="147594a9015" ut="147594a9015" version="10" userId="10059">
    <sigs count="1">
        <cert index="4" />
    </sigs>
    <signing-keyset identifier="1" />
</package>
```

Listing 4.1 – Entrée dans le fichier `packages.xml` ajoutée suite à l'installation d'une nouvelle application par un échantillon de DroidKungFu

Avant de créer les fichiers dans la partition `system`, le processus `gjsvro`

écrit dans un fichier `hotplug`. Ce fichier est une entrée du `procfs`, un système de fichier servant d'interface pour accéder à des informations sur les processus et d'autres éléments du système tels que le noyau. Si l'écriture de donnée dans le fichier `hotplug` ne signifie pas forcément une attaque, elle est cependant inhabituelle et correspond à l'exploitation de la vulnérabilité pour obtenir les droits `root` sur le système [31].

Un SFG est un multigraphe orienté qui représente de manière compacte les flux d'information observés par AndroBlare. Comme nous l'avons montré à la fin de la section 4.3, une centaine de milliers d'entrée dans le journal d'AndroBlare se réduit en un graphe avec un moyenne une centaine d'arcs. Grâce à sa compacité, cette structure facilite l'analyse des flux observés dans le système afin de comprendre le comportement d'une application. Dans le cas d'analyse de malware, cette fonctionnalité s'avère intéressant pour établir un début de diagnostic d'une attaque. Pour illustrer cela, nous avons analysé avec AndroBlare un échantillon du malware DroidKungFu1 et construit le SFG correspondant aux flux observés durant l'analyse. Nous avons déduit à partir du SFG que l'échantillon analysé installait deux applications dans le système : une native et une sous forme d'`apk`. Ces deux applications sont installées dans la partition `system`, ce qui rend leur présence persistante sur le téléphone. Un utilisateur normal ne peut désinstaller une application dans la partition `system` sans les droits `root` or ils ne sont pas disponibles par défaut sur les téléphones. Le SFG a également mis en évidence l'exploitation de la vulnérabilité pour avoir les droits `root` (écriture de données sensibles dans le fichier `hotplug`).

Revenons sur le travail effectué dans [16] et présenté en section 3.2. Le but de ce travail était de définir manuellement une politique de flux d'information pour le système Android. Sa réalisation a cependant mis en évidence la difficulté d'une telle approche. L'un des pré-requis à cette démarche est une connaissance approfondie du système. Or ce n'est pas souvent le cas pour les développeurs d'application et définir une politique de flux pour des applications censées tourner dans un environnement AndroBlare pourrait s'avèrer difficile. Avec T. Saliou, nous avons ainsi proposé dans [20] une approche semi-automatique pour assister un développeur dans la création de la politique d'une application.

4.5 Création d'une politique de flux d'information à partir d'un System Flow Graph

La difficulté dans la définition d'une politique est de connaître tous les conteneurs légaux des informations à surveiller. Lors de la définition de la politique d'une application, il s'agit donc d'identifier tous les conteneurs pouvant accéder ou stocker les données de cette application. Pour les identifier, nous proposons d'analyser les applications pour construire leur profil sous forme de SFG. À partir du SFG, nous déduisons ensuite les conteneurs légaux des données surveillées

car le SFG décrit où les informations surveillées se propagent dans le système.

La construction de la politique se fait en deux étapes. Dans un premier temps, le développeur analyse son application dans un environnement AndroBlare et construit le profil de son application comme décrit en section 4.4. Le but de cette étape est d'obtenir dans le SFG tous les flux d'information possibles que l'application peut causer. Le développeur étant celui qui connaît le mieux son application, il est le mieux placé pour stimuler son application et couvrir ainsi toutes les fonctionnalités offertes par l'application.

Une fois le SFG obtenu, nous calculons la politique de flux de l'application. L'algorithme 2 décrit le processus de calcul. Les fonctions `cont, ptag, xptag, inedges` et `bin` retournent respectivement le conteneur d'information associé à un nœud, le *ptag* d'un conteneur, son *xptag*, les arcs entrants d'un nœud et le fichier contenant l'application exécutée par un processus. Pour chaque nœud du SFG, nous considérons que le conteneur d'information qu'il représente est un conteneur légal de toutes les information sensibles impliquées dans les arcs entrants du nœud. Nous considérons que toutes ces informations peuvent se mélanger dans le conteneur. Par exemple, si deux arcs e_0 et e_1 sont les arcs entrants d'un nœud n alors nous considérons que le conteneur représenté par n est un conteneur légal des informations impliquées dans les flux décrits par les deux arcs.

Algorithme 2 : Calcul d'une politique de flux d'information Blare à partir d'un SFG

```
Input : Un SFG G
Output : Une politique de flux d'information
begin
    ptag (socket) ← ∅;
    foreach n ∈ nodes (G) do
        ids ← ∅ ;
        foreach e ∈ inedges (n) do
            ids ← ids ∪ e.flow ;
        switch type (n) do
            case file
                ptag (cont (n)) ← {ids};
                break;
            case process
                xptag (bin (cont (n))) ← {ids};
                break;
            case socket
                ptag (socket) ← ptag (socket) ∪ {ids};
                break;
```

En utilisant cette approche, nous avons construit la politique de trois applications issues de Google Play : Angry Birds, Finger Scanner et Knife Game.

La politique produite est écrite en BSPL [57] et le listing 4.2 est un extrait de l'une des politiques BSPL produites. Écrire la politique en BSPL permet de la composer avec d'autres politiques, notamment celle du système sur lequel l'application est installée. La politique liste les conteneurs d'information pris en compte par la politique, les données sensibles à surveiller et pour chaque donnée sensible la liste des conteneurs légaux ainsi que les informations avec lesquelles elles peuvent se mélanger dans le conteneur.

Évaluation des politiques produites

Nous avons mené deux types d'expérience afin d'évaluer les politiques de flux produites à partir de notre approche. La première expérience consiste à valider la prise en compte des flux que l'application engendre durant son exécution par sa politique de flux. Nous nous assurons qu'aucune alerte n'est levée par AndroBlare lorsque nous exécutons l'application et appliquons sa politique. La deuxième expérience consiste à valider la capacité de détection de la politique en cas d'intrusion dans le système. Plus précisément, nous vérifions que la politique de sécurité permet la détection des déviations par rapport au comportement d'origine de l'application. Nous considérons ici le cas des applications malveillantes qui sont à l'origine des applications bénignes mais auxquelles du code malveillant a été rajouté. Ce mode d'infection est le plus utilisée par les auteurs de malware selon l'analyse dans [113].

Dans les deux types d'expériences, nous appliquons la procédure suivante. Nous installons l'application, appliquons sa politique et l'utilisons comme un utilisateur normal l'utiliserait. En parallèle, nous vérifions avec AndroBlare si les flux causés par l'application violent la politique de flux du mise en œuvre. Lors de la première expérience, nous utilisons la version de l'application ayant servi lors de la création de la politique et vérifions qu'aucune alerte n'est levée lors de l'utilisation de l'application. Si aucune alerte n'est levée, cela signifie que la politique couvre tous les flux que l'application cause durant son exécution. Lors de la deuxième expérience nous utilisons des versions infectées des applications utilisées dans l'expérience précédente. Ces versions infectées sont des réels échantillons de malware provenant de la collection Contagio [82]. Les versions infectées d'Andry Birds, Finger Scanner et Knife Game sont respectivement des échantillons d'une variante de LeNa [104], DroidKungFu1 [61] et Bad News [91]. Chacun de ces malwares essaient d'installer des applications sur le téléphone soit en usant des privilèges demandées à l'installation soit en exploitant des vulnérabilités dans le système. Le but de la deuxième expérience est donc de détecter ces comportements.

Le tableau 4.1 liste le nombre d'alertes levées lors de l'évaluation des politiques des trois applications. Lors de la première expérience, AndroBlare n'a lévé aucune alerte pour les 3 applications. Les politiques respectives des trois applications couvrent donc tous les flux qu'elles engendrent durant son exécution. Lors de la deuxième expérience, AndroBlare a levé des alertes pour chacune des versions infectées des applications utilisées durant la première expérience. Les alertes levées étant nombreuses, nous ne les mettrons pas dans le présent

document mais la figure 4.7 est un extrait de ces alertes. Elle liste les alertes levées par AndroBlare lors de l'analyse de la version infectée de Finger Scanner.

La version infectée d'Angry Birds est un échantillon de LeNa. Les analyses sur LeNa indiquent que le malware exploitent une vulnérabilité du système afin d'obtenir les privilèges `root` puis installer une nouvelle application. Lors de l'analyse de l'échantillon, AndroBlare a levé des alertes indiquant l'accès en lecture et écriture à des fichiers dans le répertoire de l'application par d'autres applications : `logo` et `logcat`. `Logo` est une application binaire incluse par l'auteur du malware dans l'application. `Logcat` est une application qui permet de lire le contenu du journal du système. Ces deux applications lisent et écrivent dans les fichiers `logo, crashlog, flag, exec` et `.e1240987052d` situés dans le répertoire de l'application analysée. Si ces accès n'indiquent pas forcément une intrusion dans le système, elles correspondent cependant à la première étape de l'attaque menée par le malware, l'exploitation de la vulnérabilité sur le système. L'analyse effectuée dans [95] indique que l'application installe ou remplace les binaires situées dans `/system/bin`. AndroBlare n'a levé aucune alerte indiquant de tels comportements. Pour s'assurer qu'il ne s'agit pas de faux négatif, nous avons listé le contenu du répertoire et vérifié si de nouveaux binaires ont été créés ou si le contenu des fichiers présents ont été changé pendant l'analyse de l'application. Notre analyse a montré qu'aucun fichier n'a été créé ni modifié.

La version infectée de Finger Scanner est un échantillon de DroidKungFu 1. Tout comme LeNa, DroidKungFu1 exploite une vulnérabilité dans le système Android afin d'élever ses privilèges et installer deux applications sur le téléphone : une native et une application Android. Lors de l'analyse de l'échantillon de DroidKungFu 1, AndroBlare a levé des alertes indiquant l'écriture de données sensibles dans une entrée de `procfs`[3], deux fichiers dans la partition `system` et la propagation des données sensibles vers le fichier contenant la liste des applications du téléphone, l'application en charge d'une partie de l'installation des nouvelles applications sur le téléphone et un nouveau processus dont le nom est une partie du nom d'un des nouveaux fichiers créés dans la partition `system`. Ces alertes sont listées dans la figure 4.7. Ces alertes correspondent à l'exploitation de la vulnérabilité par DroidKungFu 1 ainsi que l'ajoute de nouvelles applications dans le système. La première alerte correspond à l'exploitation de la vulnérabilité par le malware. Les alertes qui restent décrivent l'ajout des deux nouvelles applications `system` ainsi que l'exécution de l'une d'entre elles (`com.google.ssearch.apk`).

La version infectée de Savage Knife Game est un échantillon de BadNews. BadNews est un malware dont le comportement est dicté par un serveur de commande. En analysant le code de l'échantillon avec Androguard [97], nous avons découvert que le malware peut recevoir X types de commande : installer une application, afficher une notification à l'utilisateur, changer l'adresse des serveurs de commande, télécharger des fichiers et ajouter des raccourcis soit vers des pages web soit vers des fichiers sur le téléphone. Lors de l'analyse de l'échantillon, AndroBlare a levé 209 alertes. Une partie d'entre elles correspondent au

3. Système de fichier servant d'interface à des données du système, y compris le noyau

```
[POLICY_VIOLATION] process gjsvro:gjsvro 984 > file /proc/sys/kernel/
        hotplug 4827 > itag[-3]
[POLICY_VIOLATION] process gjsvro:gjsvro 984 > file /system/bin/gjsvr
        16738 > itag[-3 3]
[POLICY_VIOLATION] process gjsvro:gjsvro 984 > file /system/app/
        com.google.ssearch.apk 8330 > itag[-3 3]
[POLICY_VIOLATION] process cat:cat 990 > file /system/app/com.google.
        ssearch.apk 8330 > itag[3]
[POLICY_VIOLATION] process gjsvro:gjsvro 984 > socket (127.0.0.1) 0 > itag[-3 3]
[POLICY_VIOLATION] file /system/app/com.google.ssearch.apk 8330 > process
        dexopt:dexopt 991 > itag[3]
[POLICY_VIOLATION] process dexopt:dexopt 991 > file /data/dalvik-cache/
        system@app@com.google.ssearch.apk@classes.dex 24632 > itag[3]
[POLICY_VIOLATION] file /data/dalvik-cache/system@app@com.google.ssearch.apk
        @classes.dex 24632 > process dexopt:dexopt 991 > itag[3]
[POLICY_VIOLATION] process droid.gallery3d:droid.gallery3d 995 > file /data/data/
        com.android.gallery3d/shared\_prefs/com.android .gallery3d\_preferences.xml
        57603 > itag[3]
[POLICY_VIOLATION] file /system/app/com.google.ssearch.apk 8330 > process
        .google.ssearch:.google.ssearch 1059 > itag[3]
[POLICY_VIOLATION] file /data/dalvik-cache/system@app@com.google.ssearch.apk
        @classes.dex 24632 > process .google.ssearch:.google.ssearch 1059 > itag[3]
```

FIGURE 4.7 – Extrait des alertes levées par l'échantillon de DroidKungFu1 lors de l'évaluation de la politique de Finger Scaner

	Version originale	Version infectée
Angry Birds	0	15
Finger scanner	0	11
Knife game	0	209

TABLE 4.1 – Nombre d'alertes levées par Blare lors de l'exécution des versions originales et infectées de trois applications en appliquant une politique BSPL

téléchargement de deux applications, leur installation et leur exécution sur le téléphone. L'une des applications est présentée comme étant Adobe Flash mais est en réalité un jeu et l'autre application est une version infectée du jeu Doodle Jump. L'autre partie des alertes décrivent l'échange de données sensibles du navigateur avec d'autres objets du système. Ces alertes sont dues au fait que l'échantillon analysé utilise le navigateur pour initier le téléchargement des applications. Pour cela, il émet un `intent` pour ouvrir les adresses web pointant vers les applications à télécharger. À partir ce moment, le navigateur se trouve ainsi marqué comme contenant une donnée sensible et tout échange qu'il aura avec les autres éléments du système est vu par AndroBlare comme impliquant une donnée sensible.

```
<BSPL_policy>

  <Data_policy>
     <data>
        <data_identification
           alias="fingerscanner"
           case_of_unknown_containers="Ask"
           data_type="I"
           origin="/data/app/-
              -mobi.thinkchange.fingerscannerlite-1.apk"
           reaction="alert" />

       <can_flow
           into="c_fingerscanner_0"
           mixed_with="" />
       <can_flow
            into="c_fingerscanner_1"
            mixed_with="" />
       <can_flow
            into="mediaserver"
            mixed_with="" />
       <can_flow
            into="ndroid.launcher"
            mixed_with="" />
    </data>
        ...
        ...
  </Data_policy>

   <Containers_policy>
      <container>
         <container_identification
            alias="c_fingerscanner_0"
            case_of_unknown_data="AlwaysAccept"
            container_type="File"
            origin="/data/data/mobi.thinkchange-
               .fingerscannerlite/databases/goo-
               gle_analytics.db"
            owner="mobi" reaction="alert" />

         <can_receive
            mixture_of_known_data="fingerscanner" />
       </container>
        ...
        ...

     </Containers_policy>

</BSPL_policy>
```

Listing 4.2 – Extrait de la politique BSPL de l'application Finger Scanner

Résumé

Nous avons présenté dans ce chapitre une structure de données, SFG, représentant sous la forme de multigraphe orienté les flux d'information qu'AndroBlare détecte durant l'exécution du système. Cette structure est le deuxième apport de la thèse et la deuxième étape vers l'accomplissement de notre objectif principal qui est de classifier et détecter les malware Android. Dans l'exemple proposé en section 4.4, nous avons obtenu un SFG contenant moins de 100 arcs à partir de plus de 3000 flux observés. Grâce à sa compacité, le SFG permet de comprendre rapidement les évènements qui se passent dans le système. L'analyse du graphe obtenu à partir des flux engendrés par un échantillon de DroidKungFu 1 nous a ainsi permis d'identifier l'installation de deux applications par le malware. En plus d'utiliser cette structure comme profil d'une application, nous avons également montré qu'il était possible de s'en servir pour créer la politique Blare d'une application. Nous avons testé cette approche sur trois applications populaires de Google Play et montré que les politiques produites capturaient bien les flux causées par les applications durant leur exécution et permettaient de détecter les déviations de comportement dues à tout code malveillant introduit dans les applications.

Le contenu de ce chapitre a fait l'objet de deux publications. La première a été publiée à WISG13 [18] et présente la structure SFG ainsi que la manière dont elle peut être utilisée pour comprendre le comportement d'une application. La deuxième publication a été publiée à IAS13 [20] et étendue dans un article du journal JIAS [19]. Elle présente la méthode de création de politique BSPL à partir d'un SFG ainsi que son évaluation.

Chapitre 5

Caractérisation et détection de malware Android

Nous avons présenté dans le chapitre précédent une structure de donnée qui décrit de manière compacte comment les informations surveillées par AndroBlare se propagent dans le système et proposé d'utiliser cette structure pour décrire comment une application propage ses données dans le système. En analysant le graphe d'un échantillon de DroidKungFu 1, nous avons montré qu'une partie du graphe correspondait à l'attaque effectuée par le malware. Selon l'analyse effectuée dans [113], plus de 86% des échantillons de malware Android sont des applications existantes auxquelles du code malveillant a été ajouté. Cela signifie donc que les échantillons d'un même malware ont un comportement partiellement commun dû au code malveillant qui leur a été injecté. Un échantillon d'un malware est une application considérée comme étant une échantillon de ce malware. Une partie des flux qu'ils causent devraient ainsi être les mêmes. En supposant que l'attaque observée dans le graphe de l'échantillon de DroidKungFu 1 soit dû à un code qui a été injecté dans l'application d'origine et d'autres applications, peut-on ainsi retrouver ce sous-graphe dans le graphe des autres échantillons ? Nous avons généralisé ce problème et répondons dans ce chapitre aux deux questions suivantes.

1. Existe-t-il des sous-graphes communs aux SFG des applications malveillantes telles que cette partie commune corresponde au comportement introduit par le code malveillant dans ces applications ?
2. Si un tel sous graphe existe, permet-il de détecter d'autres échantillons du malware ?

Afin de répondre à ces questions, nous présentons et évaluons dans ce chapitre une méthode pour calculer ce sous-graphe en commun et une méthode de détection utilisant les flux d'information et les sous-graphes en commun pour détecter l'exécution d'échantillon de malware.

5.1 Caractérisation de malware Android : classification d'applications malveillantes Android

Nous proposons de caractériser un malware avec le(s) sous-graphe(s) en commun aux SFG de ses échantillons qui décrit son comportement malveillant. Afin d'extraire cette partie commune, nous calculons les arcs en commun aux SFG des échantillons de malware comme décrit par l'algorithme 3. Le calcul effectué peut être vu comme un processus de classification non supervisée où nous cherchons à regrouper les SFG, implicitement les applications correspondantes, ayant des parties communes et parallèlement à calculer un profil pour chaque classe qui caractérise ses éléments. Idéalement, les échantillons d'un même malware seraient regroupées au sein d'une même classe et donc caractérisés par un seul profil. Contrairement aux autres approches effectuant des classifications non supervisées sur les échantillons de malware ou des applications en général [67, 89, 90, 22], nous n'utilisons aucune notion de distance entre chaque élément pour déterminer s'ils devraient faire partie d'une même classe. À la place, nous considérons que deux éléments, ici des SFG, font partie d'une même classe s'ils ont une partie commune non nulle.

Une classification consiste à apprendre à partir d'un jeu de données un modèle définissant comment ces données sont ou peuvent être regroupées. Ici nous souhaitons calculer des classes de SFG et les profils caractérisants les éléments de chaque classe. Ce jeu de données est représenté par la première liste donnée en paramètre de l'algorithme 3. Une deuxième liste est donnée en paramètre mais elle sert uniquement à filtrer les arcs qui pourraient faire partie des profils que nous calculons. Ce filtrage est effectué par la fonction `clean` au début de la classification. Nous donnerons plus de détail sur cet aspect plus tard.

Initialement, la classification associe une classe différente à chacun des SFG des applications. Le résultat renvoyé par l'algorithme, c'est-à-dire la classification finale, est obtenue en calculant un point fixe sur l'évolution de cette classification. Le calcul du point fixe est représenté par la boucle `while` tandis que la mise à jour de la classification est opérée par la fonction `one-step-classification` (algorithme 4). Cette fonction prend en entrée une liste de classe en paramètre et fusionne les paires de classes dont les profils ont une partie commune. En supposant ainsi qu'il y ait n classes dans la classification donnée en paramètre, elle vérifie pour toute combinaison de deux classes si leurs profils respectifs ont une partie commune non nulle. Cela se traduit par C_n^2 calculs d'intersection de deux SFG. Si c'est le cas, elle fusionne les deux classes et associe comme profil à la classe résultante cette partie commune non nulle. Aux éléments des deux classes ayant été fusionnées s'ajoutent les éléments des autres classes dont le profil contient l'intersection des profils des deux classes fusionnées. Si une classe n'a été fusionnée avec aucune autre classe, elle est ajoutée telle quelle dans la nouvelle classification.

Algorithme 3 : Calcul des parties communes de SFG d'application caractérisant son comportement malveillant et regroupement de ces SFG

Input :
$[g_0,\ldots,g_n]$ une liste de SFG
white une liste blanche de SFG
Output : $[(s_0,[g_{0_1},\ldots,g_{0_i}]),\ldots,(s_m,[g_{m_1},\ldots,g_{m_k}])]$ une liste de couple associant les sous-graphes communs aux graphes les contenant
begin
  $assoc \leftarrow \emptyset$;
  $new\_assoc \leftarrow$ [(`clean` (g_0, *white*),[g_0]),...,(`clean` (g_n, *white*),[g_n])];
  while ($assoc \neq new\_assoc$) **do**
    $assoc \leftarrow new\_assoc$;
    $new\_assoc \leftarrow$ `one-step-classification`($assoc$);
  return *assoc*;

Algorithme 4 : `One-step-classification` function

Input :
assoc une liste de SFG
Output : une liste d'association de SFG
begin
  $new\_assoc \leftarrow []$;
  tmp $\leftarrow \emptyset$;
  forall the $g_1 \in$ `keys`($assoc$) **do**
    forall the $g_2 \in$ `keys`($assoc$)$\backslash\{g_1\}$ **do**
      if $(g_1, g_2) \in tmp$ *or* $(g_2, g_1) \in tmp$ **then**
        continue;
      tmp $\leftarrow$ tmp $\cup\{(g_1, g_2)\}$;
      s $\leftarrow g_1 \sqcap g_2$;
      if $s \neq \emptyset$ **then**
        v $\leftarrow$ `value`($assoc, g_1$) + `value`($assoc, g_2$) ;
        forall the $g \in$ `keys`($assoc$)$\backslash\{g_1, g_2\}$ **do**
          if $s \sqsubseteq g$ **then**
            v $\leftarrow$ v + `value` ($assoc, g$)
        $new\_assoc \leftarrow add(new\_assoc, (s, v))$;
  return *new_assoc*;

Pré-traitement des éléments à classifier : filtrage des arcs des SFG

Nous avons mentionné précédemment que la deuxième liste donnée en entrée de l'algorithme 3 était utilisée par la fonction `clean` pour filtrer les arcs pouvant faire partie des profils que nous calculons. Filtrer les arcs pouvant faire partie des profils est une nécessité car une partie des SFG des applications Android est la même que ces applications soient bénignes ou non. Nous filtrons ainsi les arcs à prendre en compte dans les profils afin de ne pas calculer des profils génériques décrivant toute application Android au lieu de profils de malwrare décrivant leur comportement malveillant.

Cette partie commune entre les SFG des applications Android est due à la manière dont les applications sont écrites et à la manière dont l'environnement Android fonctionne. En effet, les composants des applications Android sont des composants qui étendent des classes prédéfinies dans Android, à savoir `Activity`, `BroadcastReceiver`, `Service` et `ContentProvider`. Les composants héritent ainsi des fonctionnalités de ces classes. Ainsi, ils ont tous par exemple la même référence au `ContextManager`, classe servant à demander les références des différents services du système. Cette classe effectue la requête au processus `servicemanager` par défaut. De plus, l'interaction avec le reste du système se fait souvent avec des fonctions fournies par l'API Android ce qui peut causer des flux similaires aux applications. Pour créer ou accéder aux préférences de l'application, les applications utilisent souvent la fonction `getSharedPreferences` par exemple. L'usage des préférences est conseillé dans divers cas tels que la sauvegarde des données lorsqu'une applications est mise en pause par le système. Le fichier stockant les préférences est localisé dans le répertoire local de l'application par défaut et son accès via l'API d'Android est ainsi le même pour toutes les applications.

À cause de ce caractère commun, des comportements communs sont partagés entre les applications qu'elles soient bénignes ou non. Ce qui implique qu'une partie des flux d'information qu'elles causent sont les mêmes. En calculant simplement le sous-graphe commun aux graphes des échantillons de malware sans effectuer de filtrage, nous risquerions d'obtenir un graphe en commun caractérisant n'importe quelle application Android et tout comportement malveillant sera ainsi absent de ce graphe. Pour palier à ce problème, il est ainsi nécessaire de filtrer les éléments à prendre en compte dans les sous-graphes en commun que nous calculons. Dans les expériences menées en section 5.2, nous filtrons les arcs à prendre en compte dans les profils de malware. Nous ignorons les arcs qui décrivent un flux impliquant les processus `system_server`, celui exécutant l'application de galerie d'images et les fichiers dans `/acct/uid`[1]. À ces flux s'ajoutent ceux décrits par les éléments de la deuxième liste donnée en entrée de l'algorithme 3. Les éléments de cette liste sont des SFG d'applications bénignes que nous considérons comme représentant les comportements communs que nous ne souhaitons pas être pris en compte durant le calcul des profils. Nous calcu-

1. Voir les pages manuel de acct sous Linux

lons dans la section 5.3 le résultat obtenu quand aucun filtrage n'est appliqué et montrons que les classes produites ainsi que les profils calculés sont trop génériques pour refléter un quelconque malware. Dans la section 5.3, nous montrons l'utilité de ce filtrage en répétant la classification effectuée en section 5.2.

5.2 Évaluation de la méthode de classification

5.2.1 Jeu de donnée

Afin d'évaluer notre algorithme de calcul de sous-graphe, nous proposons de l'appliquer sur 19 échantillons de malware : 5 échantillons de BadNews [91], 7 de DroidKungFu1 [61], 3 de DroidKungFu2 [60] et 5 de jSMSHider [96]. En plus de ces 19 échantillons, nous utilisons également 7 applications provenant de Google Play dont les SFG constitueront la liste blanche. Ces applications sont composés de quatre jeux (Angry birds, Little Dentist, Finger Scanner et Crazy Jump), un navigateur web (Firefox), deux utilitaires (Android Term et Busybox Free) et une application de fond d'écran (Ironman 3 live).

BadNews est un malware qui infecte les systèmes Android sous la forme d'applications légitimes. En analysant manuellement les 5 échantillons avec Androguard, nous avons déterminé que BadNews est un malware dont le comportement est dicté par un serveur de commande et contrôle (C&C). À l'exécution du code malveillant, il contacte le serveur afin d'obtenir la prochaine commande à exécuter. D'après l'analyse effectuée, il comprend plusieurs commandes : télécharger et installer une application, afficher des informations sous forme de notification (page web à visiter, mise à jour d'une application etc), installer de nouveaux icônes qui mènent vers une page web ou une application Android qui aurait été préalablement téléchargée et changer l'adresse du serveur C&C. Durant les périodes d'expérimentation que nous avons menées, le serveur était toujours actif et envoyait les mêmes commandes à chaque fois : télécharger deux applications et afficher des notifications de mises à jour à l'utilisateur qui une fois cliquées causent l'installation des applications. Les deux applications sont une version infectée de Doodle Jump et un jeu en Russe qui est présenté comme étant une version d'Adobe Flash pour Android.

DroidKungFu1 est un malware découvert en 2011 qui installe furtivement des applications sur le téléphone en exploitant des vulnérabilités du système ou grâce à la commande `su`. Les applications sont installées sur la partition `system` afin de rendre permanente leur présence sur le téléphone. Cette partition est par défaut montée en lecture seule et par défaut un utilisateur ne peut en changer le contenu. Un accès `root` est nécessaire pour cela.

DroidKungFu2 a un comportement similaire à DroidKungFu1. Il exploite également une vulnérabilité du système pour élever ses privilèges et installer des applications sur le téléphone.

jSMSHider est un malware qui installe également d'autres applications sur le téléphone de manière furtive. Contrairement aux deux malwares précédents, il n'exploite aucune vulnérabilité dans le système pour installer les applications.

À la place, son développeur a signé les échantillons du malware avec la clé ayant servie à signer les applications système dans les images non officielles d'Android. Ces images sont créées par des développeurs de la communauté Android qui développent des versions personnalisées du système. L'un des plus connus est Cyanogen Mod. Cette clé est celle qui est présente dans les dépôts du code source d'Android et est ainsi accessible à tous. Elle n'est cependant utilisée pour signer les applications dans les ROM officielles. En signant les échantillons avec cette clé, les échantillons obtiennent ainsi des droits réservés aux applications système telles que l'installation d'une application sur le téléphone.

5.2.2 Expérimentation et résultat

Analyse des applications

Afin d'obtenir les SFG des échantillons, il faut dans un premier temps les analyser pour observer comment leurs données se propagent dans le système. Nous avons utilisé l'environnement d'analyse décrit en section 3.5 : un téléphone Android faisant tourner la version 4.0 d'Android Ice Cream Sandwich auquel a été ajouté l'environnement AndroBlare, c'est-à-dire le noyau modifié ainsi que les applications en espace utilisateur.

Pour chaque application, nous répétons le processus suivant. Nous l'installons sur le téléphone, associons un identifiant unique à ses données en marquant son `apk` avec cet identifiant et l'exécutons. Nous utilisons chaque application comme un utilisateur lambda le ferait selon les fonctionnalités proposées par l'application.

En addition à cela, nous introduisons des évènements dans le système qui sont les éléments déclencheurs des codes malveillants présents dans les échantillons étudiés. Ces évènements ont été découverts en analysant le code de quelques échantillons de ces malwares. Nous présentons dans ce qui suit ces évènements déclencheurs mais invitons le lecteur à lire l'annexe A pour l'analyse détaillée ayant mené à leur découverte.

Afin d'éviter ou de retarder toute détection, les développeurs de malware ajoutent parfois des conditions à l'exécution de leur code. Le code malveillant dans BadNews ne s'exécute par exemple que lorsque le composant `MainService` ne reçoit un `intent` lui signifiant de s'exécuter. Pour éviter que le composant ne soit lancé à la réception de n'importe quel `intent`, le développeur de BadNews ajoute une vérification supplémentaire lors de la réception du message. Il vérifie que l'`intent` reçu contienne un paramètre `update` et que la valeur associée à ce paramètre vaut `true`. Nous envoyons ainsi manuellement un `intent` avec un paramètre `update` dont la valeur associée vaut `true` à ce composant afin que le code malveillant s'exécute et que nous puissions observer les flux d'information qu'il cause. Nous effectuons l'envoi grâ ce à la commande `am`, accessible à partir du shell d'Android comme le montre la commande ci-dessous.

```
$ am startservice APP_NAME/PACKAGE_NAME.AdvService \
        --ez update 1
```

Dans le cas de DroidKungFu1 et DroidKungFu2, certains échantillons ont une durée minimale d'attente avant d'exécuter le code malveillant. Cette durée est obtenue en soustrayant l'heure du système avec une date stockée dans un fichier `sstimestamp.xml` dans le répertoire local de l'application. Le listing 5.1 est le contenu du fichier avant toute modification pour un des échantillons de DroidKungFu1. En remplaçant la valeur stockée dans ce fichier par une valeur assez petite, par exemple 1, nous forçons l'exécution du code malveillant. Nous changeons ici le contenu du fichier `sstimestamp.xml` lors de l'analyse des échantillons de DroidKungFu1 et DroidKungFu2 afin de s'assurer que le code malveillant soit exécuté.

```
1 <?xml version='1.0' encoding='utf-8' standalone='yes' ?>
2 <map>
3     <long name="start" value="1410188921456" />
4 </map>
```

Listing 5.1 – Contenu du fichier `sstimestamp.xml` d'un échantillon de DroidKungFu1

Quant aux échantillons de jSMSHider, il n'est pas nécessaire d'introduire un quelconque évènement car le code malveillant est exécuté dès le lancement de l'application. Une fois les applications analysées, nous construisons les SFG correspondants et calculons les parties qui leur sont communes avec un outil implémentant l'algorithme 3. Nous présentons dans ce qui suit les résultats obtenus.

Résultats

À partir des SFG obtenus, nous avons calculé 4 profils, c'est-à-dire 4 SFG. À chacun d'entre eux est associé un sous-ensemble des SFG donnés en entrée. Le tableau 5.1 présente les résultats du calcul. La première colonne liste les échantillons utilisés. La deuxième colonne indique la famille à laquelle l'échantillon appartient selon la classification effectuée par les auteurs de la collection d'où il provient. La troisième colonne indique la collection d'où provient l'échantillon. Les colonnes restantes représentent chacune un profil qui a été calculé à partir des SFG des échantillons. Pour chaque échantillon, une case non vide dans l'une de ces colonnes signifie que le SFG de l'échantillon contient le SFG correspondant au profil. Par conséquent, le profil caractérise le comportement de l'échantillon. Les profils calculés sont illustrés par les figures 5.1, 5.2, 5.3 et 5.4. Le SFG de l'échantillon `live.photo.savanna.apk` contient par exemple S0, le SFG illustré par la figure 5.1. Sur les 19 échantillons utilisés, 17 d'entre eux sont regroupés exactement selon leur classification dans leur base d'origine.

Le profil S0, figure 5.1, caractérise les échantillons de BadNews. Il décrit l'envoi des données de l'application vers un deux serveurs distant, le téléchargement de deux applications et une partie de leur installation ainsi que de leur exécution. L'envoi des données correspond aux flux partant du navigateur, `and-`

Empreinte MD5 des échantillons	Famille†	Origine§	S0	S1	S2	S3
98cfa989d78eb85b86c497ae5ce8ca19	BN	C	√			
e70964e51210f8201d0da3e55da78ca4	BN	I	√			
4ecf985980bcc9b238af1fdadd31de48	BN	I	√			
ccab22538dd030a52d43209e25c1f07b	BN	I	√			
3a648e6b7b3c5282da76590124a2add4	BN	I	√			
994af7172471a2170867b9aa711efb0d	DKF1	G		√		
39d140511c18ebf7384a36113d48463d	DKF1	G		√		
7f5fd7b139e23bed1de5e134dda3b1ca	DKF1	A		√		
107af5cf71f1a0e817e36b8deb683ac2	DKF1	A		√		
6625f4a711e5afaee5f349c40ad1c4ab	DKF1	G		√		
6b7c313e93e3d136611656b8a978f90d	DKF1	A			√	
4f6be2d099b215e318181e1d56675d2c	DKF2	G			√	
72dc94b908b0c6b7e3cb293d9240393c	DKF2	G			√	
f438ed38b59f772e03eb2cab97fc7685	DKF2	G			√	
ac2a5a483036eab1b363a7f3c2933b51	DKF1	A			√	
0417b7a90bb5144ed0067e38f7a30ae0	JSH	G				√
a3c0aacb35c86b4468e85bfb9e226955	JSH	G				√
d25008db2e77aae53aa13d82b20d0b6a	JSH	A				√
24663299e69db8bfce2094c15dfd2325	JSH	A				√

† BN : BadNews, DKF1 : DroidKungFu1, DKF2 : DroidKunFu2, JSH : jSMSHider
§ C : Contagio, G : Genome Project, A : Androguard, I : Internet

TABLE 5.1 – Classification des 19 échantillons de malware.

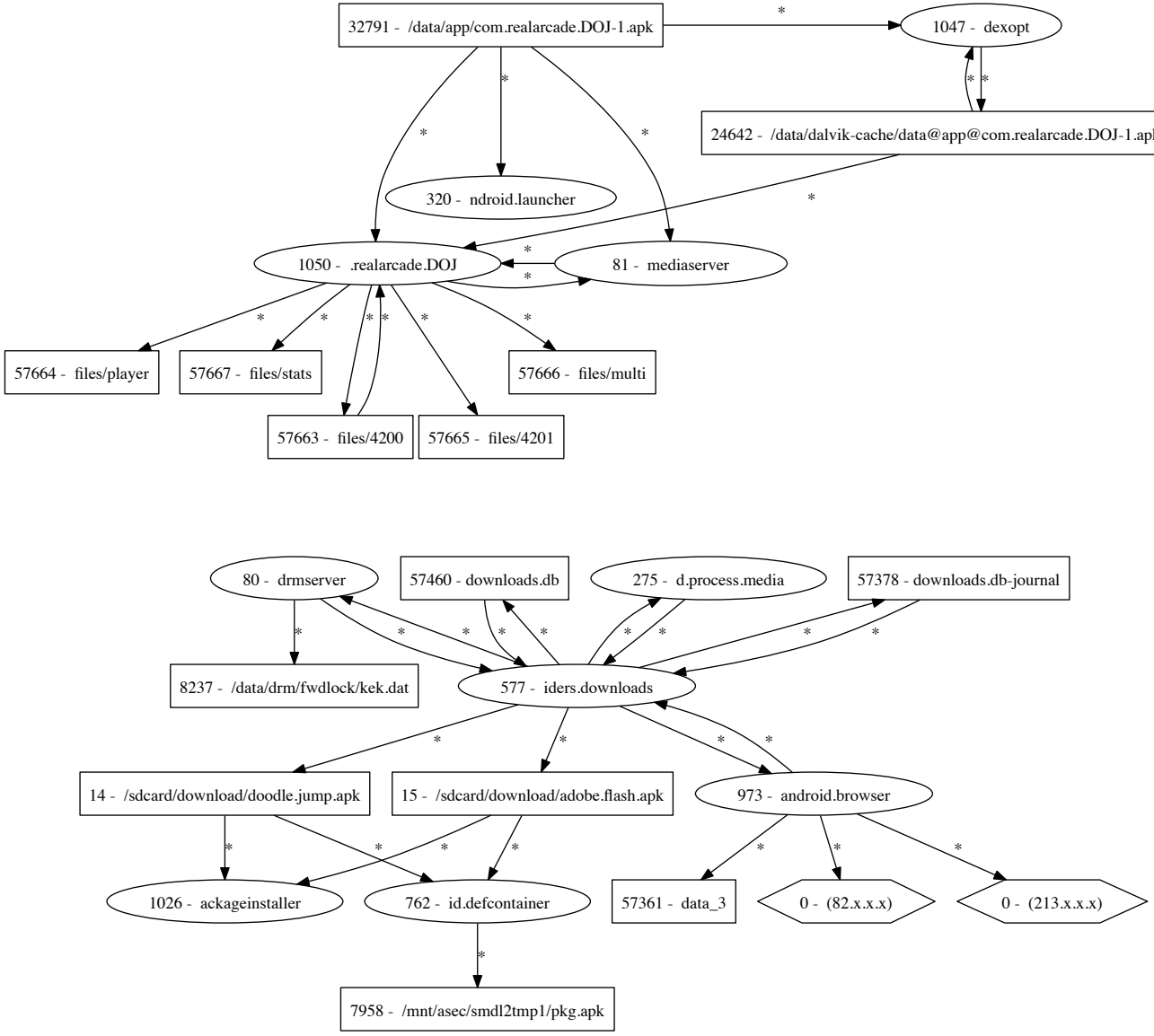


FIGURE 5.1 – S0 : sous-graphe en commun des échantillons de BadNews

`roid.browser`, vers deux sockets réseaux. Sur la figure, nous avons intentionnellement masqué une partie des adresses IP. Après vérification, l'une des adresses, `213. x.x.x`, est celle du serveur à partir duquel les applications sont téléchargées. Le téléchargement est décrit par l'écriture de données sensibles dans deux fichiers `apk` par le processus `iders.downloads`. Ce dernier exécute l'application par défaut en charge des téléchargements de fichier. L'accès à ces deux fichiers par les processus `id.defcontainer` et `packageinstaller` indique leur installation et la première composante connexe du SFG en partant du haut décrit l'exécution d'une nouvelle application, une de celles qui ont été téléchargées par le malware. Ici, il s'agit de la version infectée de Doodle Jump. L'exécution de l'autre application n'est pas présente dans le profil car il est installé dans un répertoire chiffré qui ne supporte pas les attributs étendus. Il est ainsi impossible pour AndroBlare de suivre les flux d'information impliquant les fichiers dans ce type de répertoire.

Le profil S1, figure 5.2, caractérise les échantillons de DroidKungFu1 à l'exception de deux d'entre elles que nous expliquerons dans le paragraphe suivant. S1 décrit la copie de deux fichiers dans la partition `system` et l'exécution de l'une d'entre elles par la suite. Leur destination indique qu'il s'agit d'une application native et d'une application Android.

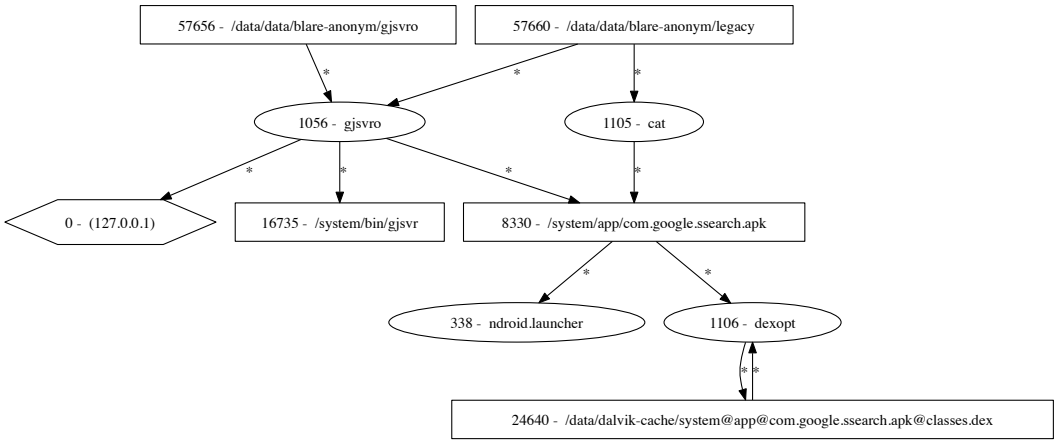


FIGURE 5.2 – S1 : sous-graphe en commun des échantillons de DroidKungFu 1

Le profil S2, figure 5.3, caractérise les échantillons de DroidKunFu2 ainsi que deux échantillons de DroidKungFu1 qui ne sont pas caractérisés par S1. S2 indique l'accès du contenu de deux fichiers au contenu teintés par les processus `secbino` et `cat`. Si ces flux n'indiquent rien de malveillant en soi, elles correspondent cependant au début de l'attaque effectuée par le malware. Ces processus servent en effet à la copie d'applications malveillantes dans le système. Comme

mentionné précédemment, deux échantillons de DroidKungFu1 sont également caractérisés par S2. La raison est que ces échantillons partagent uniquement des comportements en commun avec les échantillons de DroidKunFu2 en terme de flux. Ainsi aucune fichier `gjsvro` ou `com.google.search.apk`, ni les processus correspondants n'ont été créés durant leur analyse.

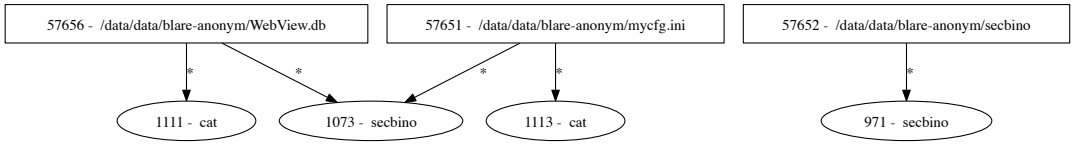


FIGURE 5.3 – S2 : sous-graphe en commun des échantillons de DroidKungFu 2

Le profil S3, figure 5.4, caractérise les échantillons de jSMSHider et décrit l'installation ainsi que le début de l'exécution d'une nouvelle application. L'accès au fichier `testnew.apk` par le processus `id.defcontainer` indique l'installation d'une nouvelle application. La lecture du fichier `jSMSHider.apk` par le processus `dexopt` puis la création d'un fichier `.dex` par ce dernier indique l'exécution d'une nouvelle application. En effet, le processus `dexopt` est celui en charge d'extraire le code des applications à partir de l'`apk` et de créer sa version optimisée.

À partir des SFG d'échantillons de 4 familles de malware, nous avons extrait 4 profils comportementaux sous la forme de SFG. Chacun de ces profils correspond à une famille de malware et décrit une partie des comportements connus de ces malwares. Le calcul de ces profils est rendu possible grâce aux SFG de 7 applications bénignes qui nous servent à filtrer les éléments faisant partie de ces profils. La liste d'application est composée de 5 jeux (Angry Birds, Crazy Jump, Fingerprint scanner et Dentist), d'un navigateur internet (Firefox) et de deux utilitaires (un terminal et la boîte à outil `busybox`).

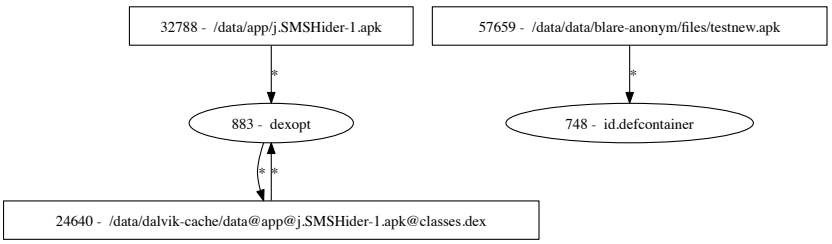


FIGURE 5.4 – S3 : Sous-graphe en commun des échantillons de jSMSHider

Filtrage	Nombre de groupe créés
Aucun filtrage	1
Sans la liste blanche	4
Filtre de base plus 7 SFG	4
20 applications de Google Play	4

TABLE 5.2 – Nombre de profils obtenus en variant le filtrage

5.3 De la nécessité du filtrage

Nous appliquons un processus de filtrage dans l'évaluation effectuée en section 5.2.2. Ce filtre est composé des arcs décrivant des flux avec les processus `system_server` et `mediaserver` ainsi que ceux des SFG de 7 applications. La raison de ce filtrage est d'éviter de calculer des classes et profils trop génériques qui ne reflètent aucun malware. Afin de montrer la nécessité de ce filtrage, nous répétons l'expérience mais en faisant varier le filtrage. Nous avons réalisé l'expérience sans utiliser aucun filtre, c'est-à-dire sans liste blanche ni les flux impliquant les conteneurs cités en section 5.1, sans la *liste blanche* puis en utilisant le filtrage de la section précédente mais en ajoutant des SFG supplémentaires à la *liste blanche*. Le tableau 5.2 présente les résultats obtenus. Chaque ligne présente le résultat obtenu selon le filtrage appliqué. La première colonne liste les différents filtrages et la seconde le nombre de profils/groupes créé. Nous discutons dans ce qui suit de la pertinence des groupes créés.

En utilisant aucun filtrage, l'algorithme extrait un seul sous-graphe qui est une partie commune aux SFG des 19 échantillons de malware utilisés. La figure 5.5 illustre ce sous-graphe. Les flux décrits par le sous-graphe indique uniquement des échanges d'information entrele processus `system_server` et différents processus. Le profil calculé est trop générique et ne décrit aucun comportement malveillant connu des malwares utilisés.

En filtrant sans la liste blanche, nous obtenons 4 profils associés à 4 groupes différents. Si le nombre de profils correspond au nombre de malwares utilisés, ils ne décrivent cependant aucune action malveillante et les échantillons dans chaque groupe sont des échantillons de malware différent dont le comportement en terme de flux d'information ne devrait pas être le même. Les figures 5.6, 5.7, 5.8 et 5.9 illustrent les profils calculés et décrivent des flux tout à fait normaux dans le système. La première figure décrit par exemple un simple échange d'information entre les applications système `com.android.phone` et `com.android.systemui`.

En étendant la liste blanche utilisée en section 5.2.2 de 20 SFG supplémentaires, nous obtenons les mêmes profils et les mêmes groupes que ceux calculés dans la section 5.2.2. Augmenter le nombre d'éléments dans cette liste est inutile et les SFG des applications qui la composent représentent de manière satisfaisante les comportements génériques que nous souhaitions ignorer durant notre classification.

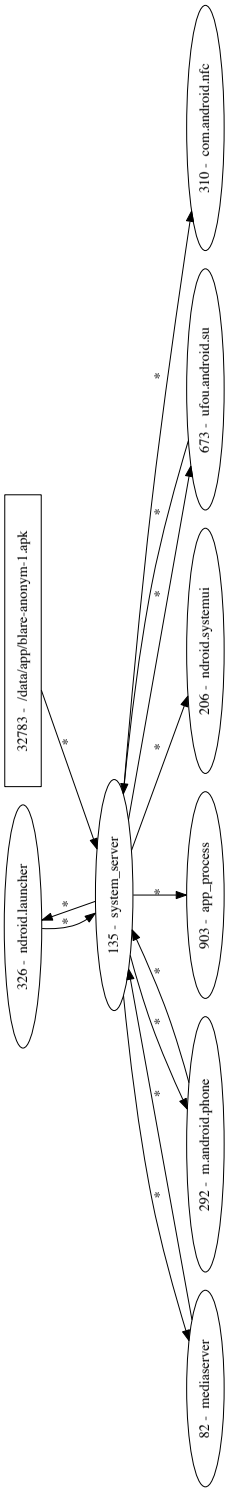


FIGURE 5.5 – Profil calculé lorsqu'aucun filtrage n'est réalisé

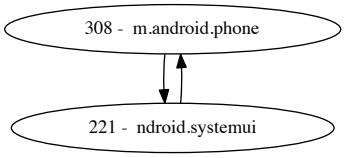


FIGURE 5.6 – Premier profil calculé en utilisant aucune liste blanche

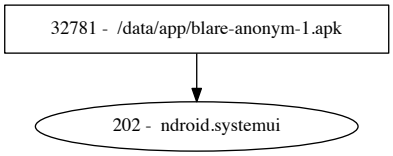


FIGURE 5.7 – Second profil calculé en utilisant aucune liste blanche

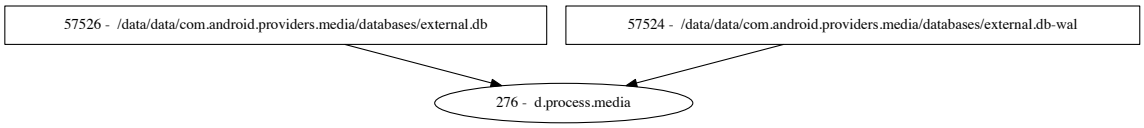


FIGURE 5.8 – Troisième profil calculé en utilisant aucune liste blanche

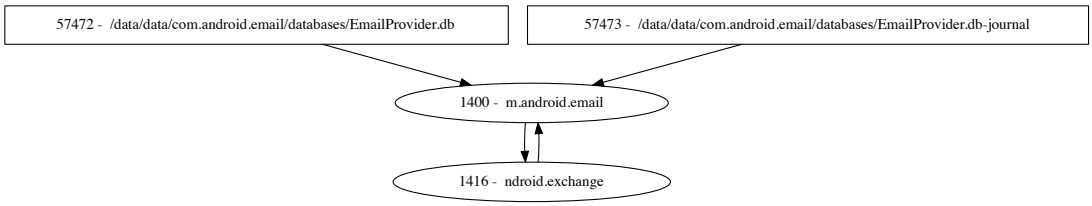


FIGURE 5.9 – Quatrième profil calculé en utilisant aucune liste blanche

L'algorithme 3 que nous avons proposé dans cette section est algorithme de classification qui regroupe les SFG des applications ayant des parties communes non nulles et calcule à la fois le profil correspondant à chaque classe. Nous avons évalué cet algorithme avec un jeu de données de 19 échantillons provenant de 4 familles de malware différentes. En appliquant l'algorithme sur les SFG des 19 échantillons, nous avons calculé 4 profils tels que chaque profil corresponde à une famille de malware différent et décrit le comportement malveillant du malware auquel il correspond. Dans la section qui suit, nous proposons une méthode de détection d'échantillon de malware en utilisant les profils calculés par notre algorithme.

5.4 Détection d'exécution de malware Android

Les résultats précédents, tableau 5.1, montrent que les échantillons de malware peuvent être caractérisés par un SFG qui décrit une partie de comportement malveillant qu'ils ont. Dans cette section, nous proposons cette fois d'utiliser ces profils SFG pour détecter l'exécution d'autres échantillons de malware. Plus précisément, nous proposons d'utiliser ces profils SFG en tant que profil de référence afin de détecter si les données d'une application sous surveillance se propagent de la même manière que celle décrite par l'un des SFG.

L'algorithme 5 décrit le processus de détection pour une application/information donnée. Il prend en entrée les flux observés par AndroBlare ainsi qu'une liste de profil SFG avec lesquels il compare les flux observés pour détecter l'exécution de malware. À l'initialisation du processus de détection, nous commençons par construire une liste d'association dans laquelle nous associons chaque arc des profils donnés en entrée avec la liste des profils SFG(s) où il est présent. Cette étape facilite la recherche d'une correspondance entre les flux observés et les arcs des profils SFG. Lors de la phase de détection, nous comparons ensuite chaque flux observé avec les arcs des différents profils SFG afin de trouver une similarité entre eux. Si une similarité est trouvée, nous levons une alerte. Nous considérons qu'il y a une similarité entre un flux d'information et un arc si les conteneurs d'information de départ et d'arrivée du flux observé sont respectivement les même que ceux décrits par les nœuds de départ et d'arrivée de l'arc et que les informations qui se propagent dans le flux observé et celui du flux décrit par l'arc sont les mêmes. Le conteneur d'information dans une entrée de Blare est le même que celui représenté par un nœud d'un SFG lorsqu'ils ont le même type et le même nom. Après avoir levé l'alerte, nous enlevons l'arc de la liste d'association afin de ne pas lever la même alerte plusieurs fois.

Algorithme 5 : Détection de l'exécution de malware Android basé sur les flux d'information causé dans le système

```
Input : L journal de Blare, sig_l liste de profils SFG
begin
    notseen ← empty association list;
    forall the g ∈ sig_l do
        forall the e ∈ edges(g) do
            if e ∉ keys(notseen) then
                Add (e, {g}) to notseen;
            else
                old ← value(e, notseen);
                Remove (e, old) from notseen;
                Add the association (e, old ∪ {g}) to notseen;
    forall the entry e ∈ L do
        if to_edge(e) ∈_light keys(notseen) then
            l ← value(to_edge(e), notseen);
            forall the g ∈ l do
                Alert that a similarity with name(g) has been found;
            Remove (to_edge(e), l) from notseen;
```

5.5 Évaluation de la capacité de détection

Afin d'évaluer la capacité de détection de l'approche présentée précédemment, nous proposons deux types d'expérience. Le premier consiste à évaluer le taux de faux positif, c'est-à-dire calculer le taux de fausses alertes levées lorsque des applications bénignes sont analysées. Le deuxième consiste à évaluer le taux de vrai positif, c'est-à-dire calculer le taux d'alertes levées lorsque des applications malicieuses sont exécutées. Pour mener ces expériences, nous proposons d'analyser des applications, bénignes et malicieuses, dans l'environnement d'analyse présenté en section 3.5 et à partir des flux observés détecter les exécutions de malware.

Jeu de données

Nous utilisons 70 des applications les plus populaires [2] dans Google Play pour la première expérience et 39 échantillons de malware provenant des 4 familles de malware utilisées dans la section 5.2.2. Nous supposons que les 70 applications sont bénignes car elles n'ont levé aucune alerte lorsque nous les avons soumises à la plateforme d'analyse de VirusTotal [14].

2. au mois de Juin 2013

Analyse des applications et détection de l'exécution de malware

Le processus d'analyse de chaque applications est le même qu'en section 5.2.2. Il se traduit par son installation dans un environnement AndroBlare, le marquage de son `apk` et son exécution. Nous utilisons chaque application comme un utilisateur normal le ferait et en parallèle, nous introduisons les évènements déclenchant le code malveillant dans les échantillons de malware. Introduire ces évènements nous assure que le code malveillant se déclenche durant l'analyse afin de déterminer si oui ou non notre approche permet de détecter l'exécution du code malveillant.

Une fois les applications analysées, nous analysons les flux observés par AndroBlare pour détecter l'exécution de malware. La détection est réalisée ultérieurement à l'analyse car nous souhaitions collecter les flux et être capables de les réutiliser plus tard. L'outil utilisé est cependant capable de faire la détection en temps réel.

Résultat de la détection

Lors de l'analyse des flux engendrés par les différentes applications, notre outil a levé des alertes pour chacun des échantillons de malware et aucune pour les applications bénignes. Les tableaux 5.3 et 5.4 résument le résultat des expériences menées respectivement sur les applications bénignes provenant de Google Play et les échantillons de malware.

Le premier tableau présente les résultats avec les applications bénignes provenant de Google Play. La première colonne liste les différentes catégories d'application utilisée durant l'expérience. La deuxième indique le nombre d'échantillons utilisés pour chaque catégorie. La troisième indique les correspondances entre les flux observés et l'une des profils utilisés. La dernière indique le total des flux observés lors de l'analyse des applications de chaque catégorie.

Le deuxième tableau présente les résultats obtenus avec les échantillons de malware. La première colonne liste les échantillons. La deuxième indique comment l'échantillon a été catégorisé dans sa collection d'origine. La troisième indique la collection d'origine de l'échantillon. La quatrième indique le nombre de flux observés par AndroBlare durant l'analyse. Les colonnes restantes indiquent le nombre de correspondance entre le profil SFG et les flux observés. Plus précisément, les valeurs dans ces colonnes indiquent le nombre d'arc du profil SFG qui correspondent à au moins un des flux observés durant l'analyse de l'échantillon.

L'analyse des flux causés par les applications bénignes montre que notre outil n'a détecté aucun flux qui correspond à l'un des arcs des profils utilisés. Sur les 70 applications bénignes analysées, aucune n'a été détectée comme exécutant du code malveillant, ce qui nous donne un taux de faux positif nul.

Chaque échantillon de malware a causé la levée d'alerte par notre outil de détection. Plus précisément, chaque échantillon a été détecté avec le profil SFG correspondant à la famille d'origine à laquelle il est supposé appartenir. Ainsi, les

Catégorie	Échantillons	Signature match	Journal entries
Jeux	48	0	7386736
Utilitaires	8	0	403427
Réseaux sociaux	5	0	567605
Photo / Vidéo	5	0	579883
Magazine	3	0	169622

TABLE 5.3 – Résultat de détection sur les applications bénignes provenant de Google Play. Taux de faux positif : 0%

échantillons de BadNews, DroidKungFu1, DroidKungFu2 et jSMSHider ont été détectés respectivement grâce au profil S0, S1, S2 et S3. La majorité des échantillons a causé autant d'alerte que de nombre d'arc que le profil pour lequel une correspondance a été trouvé. L'échantillon `live.photo.drop.apk` a par exemple causé 36 flux différents qui correspondent aux 36 arcs du profil SFG de BadNews. Cependant, quelques échantillons n'ont causé la levée d'alerte que pour une partie des arcs du profil SFG de la famille de malware à laquelle ils sont associés. Cela est dû au fait que durant leur analyse, une partie du comportement attendu ne s'est pas exécuté. L'échantillon `41f7b03a94d38bc9b61f8397af95a204` n'a par exemple causé que 4 correspondances sur 11 entre les flux qu'il a engendrés et les arcs de S1. Le profil S1 décrit l'installation de deux applications, une native et une `apk` dans la partition `system` du téléphone. L'échantillon n'a cependant installé que l'application native durant notre analyse car l'`apk` censé être installé est un fichier vide et la procédure d'installation est ainsi avortée par le système.

Empreinte MD5 des échantillons	Label†	Origin§	Log size	S0‡	S1‡	S2‡	S3‡
d25008db2e77aae53aa13d82b20d0b6a	JSH	A	79598				4/4
24663299e69db8bfce2094c15dfd2325	JSH	A	179608				4/4
39d140511c18ebf7384a36113d48463d	DKF1	A / G	3565		11/11		
7f5fd7b139e23bed1de5e134dda3b1ca	DKF1	A	5772		11/11		
a81dc5210b3444b8e6f002605a97292d	DKF1	A	3233		3/11		
107af5cf71f1a0e817e36b8deb683ac2	DKF1	A	7257		11/11		
ac2a5a483036eab1b363a7f3c2933b51	DKF1	A	3596			5/5	
e741a9bc460793b9afdadc963d6e8c1d	DKF1	A	3230		3/11		
6b7c313e93e3d136611656b8a978f90d	DKF1	A	7740			5/5	
389b416fb0f505d661716b8da02f92a2	JSH	G	179702				4/4
a3c0aacb35c86b4468e85bfb9e226955	JSH	G	7527				4/4
0417b7a90bb5144ed0067e38f7a30ae0	JSH	G	32145				4/4
d25008db2e77aae53aa13d82b20d0b6a	JSH	G	122951				4/4
f0fcef1c52631ae36f489351b1ba0238	JSH	G	211823				4/4
06dea6a4b6f77167eaf7a42cb9861bbe	DKF1	G	72706		6/11		
994af7172471a2170867b9aa711efb0d	DKF1	G	13959		11/11		
107af5cf71f1a0e817e36b8deb683ac2	DKF1	G	187221		11/11		
71fe80d5bf6d08890de3c76a3292fc09	DKF1	G	15709		11/11		
ecc4aad77ab042a4fa1693fc77afb8ac	DKF1	G	107910		11/11		
b763bc07f641bb915a4e745f1deff315	DKF1	G	180984		8/11		
6625f4a711e5afaee5f349c40ad1c4ab	DKF1	G	4982		11/11		
5c593a7ab5e61f76d2e0e61c870da986	DKF1	G	99363		11/11		
41f7b03a94d38bc9b61f8397af95a204	DKF1	G	13474		4/11		
f438ed38b59f772e03eb2cab97fc7685	DKF2	G	34906			5/5	
4f6be2d099b215e318181e1d56675d2c	DKF2	G	283990			5/5	
805bbc6ff9ef376c4b5f2c1b1c1006d2	DKF2	G	49404			5/5	
13a491126dd11f1ef51a4b067f10f368	DKF2	G	278425			5/5	
72dc94b908b0c6b7e3cb293d9240393c	DKF2	G	294163			5/5	
e4d348e97db481507a0cea64232c8065	DKF2	G	64616			5/5	
47ffc035dd1288bad27b3681535e68c8	BN	I	298819	36/36			
d8943ed5be382c22c9a206af0815ff0a	BN	I	363578	36/36			
ccab22538dd030a52d43209e25c1f07b	BN	I	167837	36/36			
3a648e6b7b3c5282da76590124a2add4	BN	I	332519	36/36			
4ecf985980bcc9b238af1fdadd31de48	BN	I	125167	36/36			
5b08c96794ad5f95f9b42989f5e767b5	BN	C	132846	36/36			
422d1290422ebfbf48ec34f0990fba21	BN	I	634202	35/36			
98cfa989d78eb85b86c497ae5ce8ca19	BN	C	568920	36/36			
e70964e51210f8201d0da3e55da78ca4	BN	I	253320	36/36			
8b9e8a2e93c3f3c18b8f5820f21e2458	BN	I	149248	36/36			

† BN : BadNews, DKF1 : DroidKungFu1, DKF2 : DroidKunFu2, JSH : jSMSHider
§ C : Contagio, G : Genome Project, A : Androguard, I : Internet

TABLE 5.4 – Résultats de la détection sur 39 échantillons de malware. Taux de Vrai Positif : 100%

Bilan et discussion

Nous avons présenté et évalué dans ce chapitre une nouvelle méthode de classification et de détection des échantillons de malware basée sur les flux d'information qu'ils causent dans le système décrits par leur SFG. La méthode de classification proposée peut-être vue comme un apprentissage non supervisé où nous cherchons à construire des classes d'échantillons de malware où les SFG des éléments d'une même classe partagent tous une partie commune. Cette partie commune est censée décrire l'attaque menée par le(s) malware(s) et caractériser les échantillons de la classe qu'elle représente en tant que profil comportemental. Nous avons évalué la méthode proposée en classifiant 19 échantillons de 4 malwares différents. La classification de ces échantillons a donné en sortie 4 classes distinctes qui représentent chacun un des 4 malwares et la partie commune des SFG des éléments de chaque classe décrit une partie de l'attaque menée par les mawares.

À partir des profils caractérisant chaque classe, nous avons proposé de détecter des échantillons des 4 malawres utilisés durant la classification. La méthode de détection consiste à comparer les flux observés causés par les échantillons analysés avec les arcs des profils comportementaux calculés. À chaque fois qu'une correspondance est trouvée, une alerte est levée. Nous avons évalué cette méthode avec 36 échantillons de malware ainsi que 70 applications bénignes provenant de Google Play et avons obtenu un taux de vrai positif de 100% ainsi qu'un taux de faux positif de 0%. Les échantillons de malware ont bien été détecté et aucune alerte n'a été levée dans le cas des applications bénignes. De plus, chaque échantillon de malware a été détecté grâce au profil caractérisant la classe associée au malware obtenue durant la classification.

La raison de la qualité des résultats obtenus aussi bien pour la classification que pour la détection est que le profil calculé ne se limite pas aux processus exécutant les échantillons de malware. Il inclut également tout élément du système ayant accès aux informations provenant des échantillons de malware. Même si un malware change son aspect, le profil calculé ne sera que très peu impacté car le malware n'a pas de contrôle ou alors très peu sur tous les autres éléments du système ayant accès à ses informations au début de l'attaque. Si le cas contraire s'avérait être vrai, c'est-à-dire que le malware a un contrôle total sur tous les autres éléments du système, alors il pourrait propager ses informations de manière différente à chaque exécution et empêcher notre algorithme de calculer un profil ou de détecter l'exécution de l'un de ses échantillons.

L'approche proposée se base sur une analyse dynamique des applications afin d'observer les flux d'information qu'elles causent dans le système. Comme toute approche dynamique, elle souffre ainsi de la même limitation qui est la couverture de code durant l'analyse et qui, pour le moment, limite la possibilité d'automatiser entièrement tout le processus d'analyse et de classification des applications. En effet, durant l'analyse nous stimulons les applications en interagissant avec elles via leur interface graphique et introduisons des évènements dans le système afin de déclencher le code malveillant dans les échantillons de malware. Cette stimulation est réalisée manuellement. De plus, il nous a nécessité

d'analyser quelques échantillons de malware afin de déterminer les évènements déclenchant le code malicieux présents dans leur code. Il serait ainsi intéressant pour stimuler les applications et couvrir le maximum de code à l'exécution.

La méthode de caractérisation de malware Android proposée dans ce chapitre a fait l'objet d'une publication à la conférence NSS 14 [17].

Chapitre 6

Conclusion

Dans cette thèse nous avons proposé et évalué une nouvelle méthode afin de caractériser / classifier et détecter les malwares Android. Afin d'y arriver, nous sommes passés par plusieurs étapes qui ont consisté à porter un moniteur de flux d'information sur Android, proposer une structure compacte et humainement compréhensible des flux d'information que le moniteur observe qui puisse aider un analyste à comprendre le comportement d'une application et classifier ainsi que détecter les échantillons de malware à partir de cette structure.

La première étape de la thèse a consisté à créer AndroBlare, une version Android de Blare. Blare a été initialement développé pour les systèmes Linux, et plus précisément pour les noyaux Linux. Si le système Android est proche des systèmes Linux, le noyau Android étant basé sur le noyau Linux et suit la même numérotation, il possède cependant quelques fonctionnalités qui lui sont propres et qu'il a fallu prendre en compte lors du portage de Blare vers Android. Ces fonctionnalités sont l'exécution des applications Android et le mécanisme de communication entre processus via le Binder. Nous avons ainsi introduit deux extensions dans AndroBlare pour prendre en compte ces deux fonctionnalités d'Android. Ces extensions ajoutent la prise en compte des flux d'information s'opérant via le Binder et un mécanisme de coopération entre les instances de la machine virtuelle Dalvik et KBlare. Ces deux extensions sont importantes car elles permettent d'observer les flux d'information entre les applications Android ainsi que l'exécution des applications. Différents mécanismes sont utilisés par les applications pour communiquer avec le reste des applications sur le système et tous ces mécanismes reposent sur le Binder. Ces communications n'étaient cependant pas visibles avec la version Linux de Blare car Binder est spécifique à Android et n'est donc pas pris en compte. La première extension comble donc cette limitation de Blare en interceptant les émissions et réception de transaction et en effectuant les opérations de propagation et de contrôle des flux d'information correspondantes. Quant aux applications Android, leur exécution est également invisible avec la version Linux de Blare. Le code de ces applications étant livré sous forme de *dalvik bytecode*, elles ne sont pas exécutées directement mais interprétées par la machine virtuelle Dalvik. La deuxième extension permet

donc, grâce au mécanisme de coopération, de combler cette autre limitation de la version Linux de Blare. Grâce à ce mécanisme de coopération, les instances de la machine virtuelle Dalvik notifient KBlare dans le noyau à chaque fois que le code d'une application est sur le point d'être interprété. À la réception d'une notification, KBlare met à jour le label du processus exécutant la machine virtuelle ayant émis la notification pour signifier le code qu'il exécute et appliquer la politique de flux d'information de l'application au processus.

La deuxième étape a consisté à proposer une structure représentant de manière plus compacte et plus compréhensible les flux d'information observés par Blare que nous avons appellé graphe de flux système ou SFG. Pour une information donnée, le SFG décrit comment elle est propagée dans tout le système. Comme nous l'avons montré dans le chapitre 4, cette structure s'avère utile sur trois aspects. Premièrement, sa représentation est plus compacte et facilite l'analyse des flux d'information observés dans le système. À chaque fois qu'un flux d'information impliquant une donnée sensible, comprendre une donnée à laquelle un identifiant a été associé, est observé, Blare ajoute une entrée décrivant ce flux dans un journal. Or comme nous l'avons montré dans le chapitre 4, certains de ces flux sont répétés plusieurs fois durant l'analyse de l'application ce qui au bout de quelques entraîne la création de milliers d'entrées dans le journal de Blare alors qu'en réalité seule une centaine de flux d'information uniques ont été observés. Ensuite, le SFG aide à comprendre le comportement d'une application. Dans notre cas, nous nous sommes intéressés aux applications malveillantes. L'analyse d'un échantillon de DroidKungFu1 a montré qu'une partie de son SFG décrivait l'attaque menée par le malware, à savoir tentative d'élévation de privilège et installation de deux applications dans le système.

La troisième et dernière étape a consisté consiste en la classfication et détection de malware. Dans cette étape, nous avons proposé d'utiliser le profil sous forme de SFG des applications afin d'extraire le profil d'un malware. Dans l'étape précédente, nous avons montré que une partie du SFG de l'échantillon de DroidKungFu1 correspondait au comportement malveillant du malware. D'après les travaux de Zhou et al. dans [113], les développeurs de malware ajoutent souvent leur code malveillant dans plusieurs applications existantes pour infecter les utilisateurs. Cela signifie donc que le même code malveillant peut être retrouvé dans les échantillons d'un même malware ou qu'une similarité en matière de comportement peut être retrouvé dans le code malveillant de ces échantillons. Partant de ce constat, nous avons émis l'hypothèse que si une partie du SFG d'une application malveillante correspond au comportement malveillant et que si la méthode d'infection principale des développeurs de malware était d'injecter leur code malveillant dans des applications existantes alors une partie des SFG des échantillons d'un même malware devrait être le même.

Direction pour des travaux futurs

Le premier apport de la thèse est AndroBlare, le résultat du portage de Blare sous Android. AndroBlare suit les flux d'information en considérant les

objets du système comme des boîtes noires. Comme nous l'avons évoqué en section 2.6, une observation au niveau système offre une vue complète des flux entre les différents objets du système mais a une granularité moins fine qu'une observation au niveau applicatif. Par exemple, dans le cas d'AndroBlare, un flux d'un objet A vers un objet B signifie que toutes les informations dans A se propagent vers B (pire des cas). Cela n'est pas forcément le cas et il se peut qu'aucune information sensible ne se soit propagée. La granularité d'observation n'est donc pas assez fine. Un axe à explorer serait ainsi de combiner les niveaux d'observation système et applicatif sous Android afin d'affiner la granularité des flux observés. L'apport d'une telle est non négligeable dans le cas de certains processus tels que `system_server` car ce processus exécute plusieurs services avec qui les applications interagissent tous, ce qui est une source de l'explosion des marques dans la version actuelle d'AndroBlare.

Un autre apport de la thèse, qui est le principal, est l'usage des SFG en tant que profil des malware. Pour calculer ce profil, nous calculons les intersections des SFG des échantillons de malware. Cette intersection n'autorise aucune déviation, même moindre, dans les flux observés. Par exemple, les flux décrivant un processus $P1$ communiquant avec un processus $P2$ via un fichier et les flux décrivant les mêmes processus $P1$ et $P2$ communiquant cette fois de manière directe sont considérés comme des flux différents. Cependant, dans la réalité, ils décrivent la même action qui est la communication entre deux processus. Nous parlons dans ce cas de flux similaire. Il serait ainsi intéressant d'explorer l'usage de la similarité à la place d'une correspondance stricte entre les flux d'information observés. L'usage de la similarité pourrait par exemple révéler la ressemblance entre deux ou plusieurs malware. Il pourrait également améliorer la résistance de l'approche face à des déviations de comportement dans les échantillons des malware.

L'approche que nous proposons pour classifier et détecter les malware est une approche dynamique. Nous analysons dans un environnement AndroBlare les applications afin de capturer les flux d'information qu'ils causent dans le système. L'une des limites de cette approche est la couverture de code. Nous couvrons durant l'exécution qu'une partie de tout le code de l'application que nous analysons. Dans le cas de l'analyse des échantillons de malware, nous avons par exemple du analysé statiquement leur code afin de trouver les évènements déclenchant le code malveillant. Ces évènements ont ensuite été introduits durant l'analyse des échantillons avec AndroBlare afin de s'assurer que leur comportement malveillant soit capturé. Quant aux applications bénignes, nous les avons utilisé comme un utilisateur normal pendant cinq minutes mais cela ne garantie pas une couverture optimale du code. Un axe à explorer serait ainsi une méthode pour couvrir tout le code des applications durant leur analyse avec AndroBlare afin que tout leur comportement soit capturé et d'automatiser tout le processus d'analyse des applications. Cela ouvrirait également la voie à une expérimentation à plus grande échelle des méthodes de classification et de détection que nous avons proposé dans cette thèse.

Il est souvent plus difficile de finir que de commencer.

Annexe A

Analyse d'échantillons de malware : recherche des évènements déclenchant leur code malveillant

Lors de la validation expérimentale des approches pour la caractérisation et la détection de malware Android, nous avons utilisé des échantillons de 4 malwares différents : DroidKungFu1, DroidKungFu2, jSMSHider et BadNews. Le but des expériences menées était de montrer qu'il était possible de caractériser sous forme de SFG le comportement malveillant des malware et d'utiliser ces SFG pour détecter l'exécution d'autres instances des malware que nous supposions non-connues. Pour capturer et détecter ce comportement malveillant, ils nous a fallu introduire des évènements dans le système afin de déclencher le code malveillant. Nous présentons dans ce qui suit l'analyse de quelques échantillons de malware afin de déterminer ces évènements.

A.1 Outils

Pour analyser les échantillons de malware, nous utilisons principalement les outils Androguard et apktool. Nous nous servons principalement d'Androguard afin de décompiler à la volée le code des applications Android et d'apktool pour extraire le contenu des fichiers `apk`, décoder en clair le contenu des fichiers `AndroidManifest.xml` ainsi que pour désassembler en `smali` le code des applications.

A.2 BadNews

BadNews est un malware Android découvert en 2013 qui se répandait sous-forme d'applications existantes infectées par un code malveillant. Le rapport d'analyse effectué dans [91] indique que le comportement de BadNews est dicté par un serveur C&C. Lorsqu'un client se connecte, le serveur lui envoie une commande à exécuter et attend sa prochaine connexion pour lui indiquer la prochaine action à effectuer. Trouver l'évènement déclencheur du comportement malveillant des échantillons de BadNews revient ainsi à trouver l'évènement qui déclenche le connexion vers le serveur puis le traitement de la commande à exécuter.

Pour identifier les évènements déclencheurs du code malveillants de BadNews, nous avons analysé l'un de ses échantillons [1] provenant de la collection Contagio. L'approche que nous avons adopté est de partir des différents points d'entrée de l'application afin d'identifier le code implémentant le comportement de BadNews, puis déterminer l'évènement attendu par l'application pour déclencher l'exécution du code. Les composants sont les points d'entrée d'une application Android. À quelques exceptions près, ils doivent tous être listés dans le fichier `AndroidManifest.xml`. Le listing A.1 présente le contenu de ce fichier pour l'échantillon analysé. Il indique que l'application possède cinq composants : deux de type `Activity`, un `Service` et deux de type `Receiver`.

L'analyse du code des deux composants `Receiver` indique qu'ils exécutent tous deux la même action à la réception d'un `intent`, listing A.2, à savoir le lancement du composant `AdvService`. L'analyse du code de `AdvService` s'avère fructueuse car parmi les méthodes de ce composant se trouve une méthode `sendRequest`, listing A.3, qui implémente le comportement attendu de BadNews : envoi d'une requête vers un serveur distant et exécution de différentes actions selon la réponse reçue : installation d'application, ajout de raccourci sur le téléphone, affichage de notification etc.

Le code ayant été identifié, il reste à déterminer comment il est appelé et sous quelles conditions. L'une des fonctionnalités d'Androguard est de construire le graphe d'appel des fonctions d'une application et de l'exporter sous un format de graphe reconnu par des outils de visualisation et manipulation de graphe tels que Gephi. Grâce à la commande `androgexf` dans Androguard, nous créons ainsi un graphe sous le format `gexf` décrivant les appels de fonction dans l'échantillon analysé et que nous pouvons ouvrir avec Gephi. La figure A.1 est un extrait du graphe d'appel produit quand nous le visualisons sous Gephi. Chaque nœud représente une méthode dont le label comporte le nom de la classe où la méthode est déclarée, son nom et sa signature. Quant aux arcs, ils représentent les appels de fonction entre les méthodes. Un arc de A vers B signifie que la méthode A appelle la méthode B. Ainsi la méthode `sendRequest` est directement appelée par `getUpdate`. En analysant la première composante connexe du graphe à partir du bas de la figure, nous remarquons que `sendRequest` est uniquement appelé durant l'exécution d'un `Thread` implémenté par la classe `AdvService$1` [2]. La com-

1. SHA sum : f1b351d1280422c5d1e3d2b1b04cb96a5d195f62
2. méthode `run`

```
19      <application android:label="@string/app_name" android:icon="@drawable/icon"
20          android:name=".App" android:allowBackup="true" android:largeHeap="true">
21          <activity android:theme="@*android:style/Theme.NoTitleBar.Fullscreen"
22              android:name=".GameActivity" android:screenOrientation="portrait"
23              android:configChanges="keyboard|keyboardHidden|orientation|
24                      screenLayout|uiMode|screenSize|smallestScreenSize"
25              android:windowSoftInputMode="stateHidden">
26              <intent-filter>
27                  <action android:name="android.intent.action.MAIN" />
28                  <category android:name="android.intent.category.LAUNCHER" />
29              </intent-filter>
30          </activity>
31          <activity android:name="com.google.ads.AdActivity"
32              android:configChanges="keyboard|keyboardHidden|orientation|
33              screenLayout|uiMode|screenSize|smallestScreenSize"/>
34          <meta-data android:name="mobidisplay_key"
35              android:value="16ab8bfc9aee9a9171ee890c8519576a" />
36          <service android:name="com.mobidisplay.advertsv1.AdvService"
37              android:exported="true" android:process=":remote">
38              <intent-filter>
39                  <action android:name="com.mobidisplay.advertsv1.AdvService" />
40              </intent-filter>
41          </service>
42          <receiver android:name="com.mobidisplay.advertsv1.BootReceiver"
43              android:enabled="true">
44              <intent-filter>
45                  <action android:name="android.intent.action.BOOT_COMPLETED" />
46                  <action android:name="android.intent.action.PHONE_STATE" />
47              </intent-filter>
48          </receiver>
49          <receiver android:name="com.mobidisplay.advertsv1.AReceiver"
50              android:enabled="true" />
51      </application>
```

Listing A.1 – Fichier `AndroidManifest.xml` d'un échantillon de BadNews

posante connexe en haut de la figure nous révèle également que le `thread` n'est instancié que lors de l'exécution de la méthode `onStartCommand` de `AdvService`, c'est-à-dire lorsqu'`AdvService` reçoit un `intent` lui demandant de s'exécuter. L'analyse du code de `onStartCommand`, listing A.4 nous le confirme et nous indique même la condition pour que le `thread` soit lancé. En effet pour qu'il soit exécuté, il faut que l'`intent` reçu par `AdvService` contienne un paramètre `update` dont la valeur doit être différente de 0 [3].

Ainsi pour lancer l'exécution du code malveillant de BadNews, il faut envoyer un `intent` avec un paramètre `update` dont la valeur est à `true` au composant `AdvService`. Dans le cas de certains échantillons, le composant AdvService est nommé différemment. Il suffit dans ce cas d'envoyer l'`intent` à ce composant. La commande `am` permet l'envoi de l'`intent` à partir de la ligne de commande sous Android comme illustré ci-dessous. Il est nécessaire de préciser à la fois le

3. Les booléens sont considérés comme des entiers sous Dalvik

nom complet de l'application ainsi que celui du composant afin que le système puisse déterminer le destinataire exact du message.

```
$ am startservice APP_NAME/PACKAGE_NAME.AdvService \
        --ez update 1
```

```
1 public void onReceive(android.content.Context p4, android.content.Intent p5) {
2     v0 = new android.content.Intent();
3     v0.setAction("com.mobidisplay.advertsv1.AdvService");
4     v0.putExtra("update", 1);
5     p4.startService(v0);
6      return;
7 }
```

Listing A.2 – Code appelé à la réception d'un `Intent` par les composants de type `Receiver` d'un échantillon de BadNews

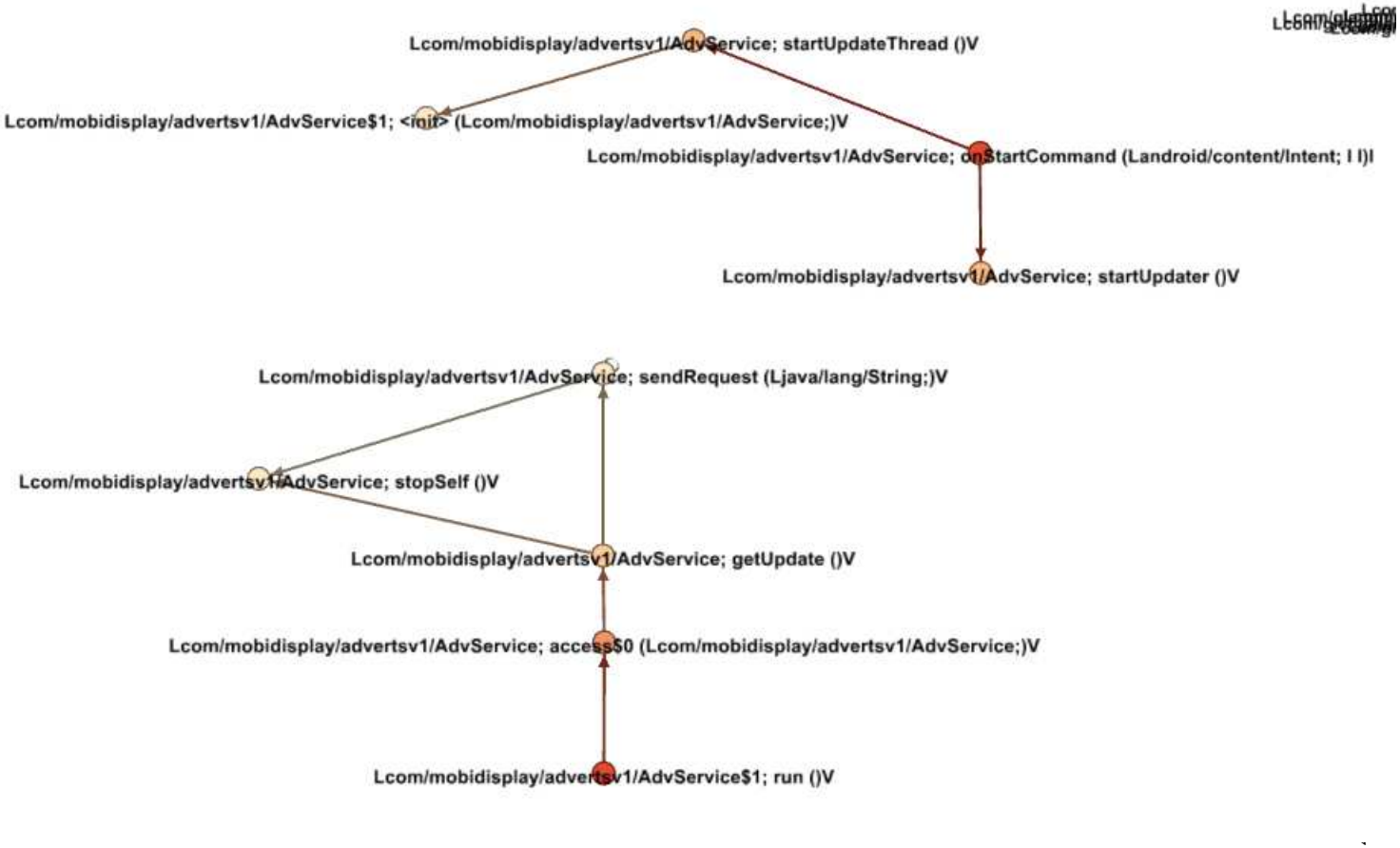


FIGURE A.1 – Extrait du graphe d'appel de fonction d'un échantillon de BadNews

```
private void sendRequest(String p24) {
        v12 = android.net.Proxy.getDefaultHost();
        v13 = android.net.Proxy.getDefaultPort();
        v4 = new java.net.URL(p24);
        if(v13 <= 0) {
            v5 = v4.openConnection();
        } else {
            v5 = v4.openConnection(new java.net.Proxy(java.net.Proxy$Type.HTTP,
                    new java.net.InetSocketAddress(v12, v13)));
        }
        v5.setDoInput(1);
        v5.setDoOutput(1);
        v5.setRequestMethod("POST");
        v5.setConnectTimeout(10000);
        this.tmett = (System.currentTimeMillis() / 1000.0);
        v18 = new java.io.DataOutputStream(v5.getOutputStream());
        v18.writeBytes(new StringBuilder("kurok=").append(this.ii)
                .append("&taket=").append(this.pacNme).append("&phone=")
                .append(this.phoNum).append("&vesn=").append(this.vesn)
                .append("&sysname=").append(this.syNm).append("&operator=")
                .append(this.opNm).append("&sdk=")
                .append(android.os.Build$VERSION.SDK).append("&tmett=")
                .append(this.tmett, 1000.0).append("&model=").append(this.phMl)
                .append("&lg=").append(this.lg).append("&io=").append(this.io)
                .append("&qyNm=").append(this.qyNm).append("&devicesid=")
                .append(android.provider.Settings$Secure.getString(this
                        .getBaseContext().getContentResolver(), "android_id"))
                .toString());
        v18.flush(); v18.close(); v5.connect();
        v6 = v5.getInputStream();
        if(v6 != 0) {
            this.increaseQueryNum();
            v19 = new java.io.InputStreamReader;
            v19(v6, "UTF-8");
            v14 = new java.io.BufferedReader(v19);
            v3 = new StringBuilder();
            while(true) {
                v8 = v14.readLine();
                if(v8 == 0) break;
                v3.append(v8).append("\x0a");
            }
            v16 = new org.json.JSONTokener;
            v16(v3.toString());
            v7 = new org.json.JSONObject(v16);
            v15 = v7.getString("status");
            if(v15.equalsIgnoreCase("news") != 0) {
                this.parseNews(v7);
            }
                        if(v15.equalsIgnoreCase("install") != 0) {
                this.parseInstall(v7);
            }
            ...
```

Listing A.3 – Extrait du code de la méthode `sendRequest` d'un échantillon de BadNews

```
public int onStartCommand(android.content.Intent p3, int p4, int p5) {
    this.log("AdvService Started");
    if(p3 == 0) {
        this.startUpdater();
    } else {
        if(p3.getExtras() == 0) {
            this.startUpdater();
        } else {
            if(p3.getExtras().getBoolean("update") != 0) {
                this.startUpdateThread();
            }
        }
    }
    return 1;
}

private void startUpdateThread() {
    new Thread(new com.mobidisplay.advertsv1.AdvService$1(this)).start();
    return;
}
```

Listing A.4 – Code appelé à l'exécution du composant `AdvService` d'un échantillon de BadNews

A.3 DroidKungFu1

DroidKungFu1 [61] est un malware découvert en 2011. Certains de ses échantillons n'exécutent pas automatiquement le code malveillant dès le lancement de l'application dans laquelle ils se trouvent. Nous avons ainsi analysé l'un des échantillons de ce malware [4] afin de trouver l'évènement déclenchant le code malveillant.

Selon l'analyse qui en a été réalisée par X. Jiang dans [61], le code malveillant du malware est stocké dans un `service` qui est ajouté à des applications existantes pour infecter les utilisateurs. Les captures effectuées dans le rapport montrent que le `service` en question s'appellerait `com.google.searchservice` et que l'envoi de données sensibles vers l'extérieur et l'installation de nouvelles applications se font respectivement grâce aux fonctions `doSearchReport` et `cpLegacyRes`. L'analyse du fichier `AndroidManifest.xml`, listing A.5, confirme qu'un tel service existe dans l'application et une analyse rapide du code de ce composant indique la présence de ces deux méthodes. Il reste ainsi à déterminer comment ces méthodes sont appelées.

Comme dans le cas de BadNews, nous construisons le graphe d'appel de fonction de l'application pour identifier comment ces méthodes sont appelées. La figure A.2 est un extrait du graphe d'appel et indique que les deux méthodes sont appelées durant la création du composant `SearchService`, plus précisément par la méthode `onCreate` de ce composant. L'analyse du code de cette méthode, listing A.6, indique que l'exécution de l'attaque ne se fait qu'après un laps de temps. À la création du composant, l'application récupère la valeur du paramètre `start` dans ses préférences et la compare avec le temps du système. Si la différence est inférieure à $14400000.0ms$ alors l'attaque n'est pas lancée et la valeur de `start` est initialisée au temps du système. Dans le cas contraire, l'attaque est lancée. La condition pour que l'attaque soit exécutée est ainsi qu'au moins $14400000.0ms$ se soit écoulé depuis le changement de la valeur du paramètre `start`. Ce paramètre est stocké dans le fichier `sstimestamp.xml`, voir listing A.7, situé dans un sous-répertoire du répertoire local de l'application. En modifiant sa valeur, nous pouvons ainsi faire croire à l'application que plus de $14400000.0ms$ s'est écoulé et forcer l'exécution du code malveillant.

4. MD5 : 7f5fd7b139e23bed1de5e134dda3b1ca

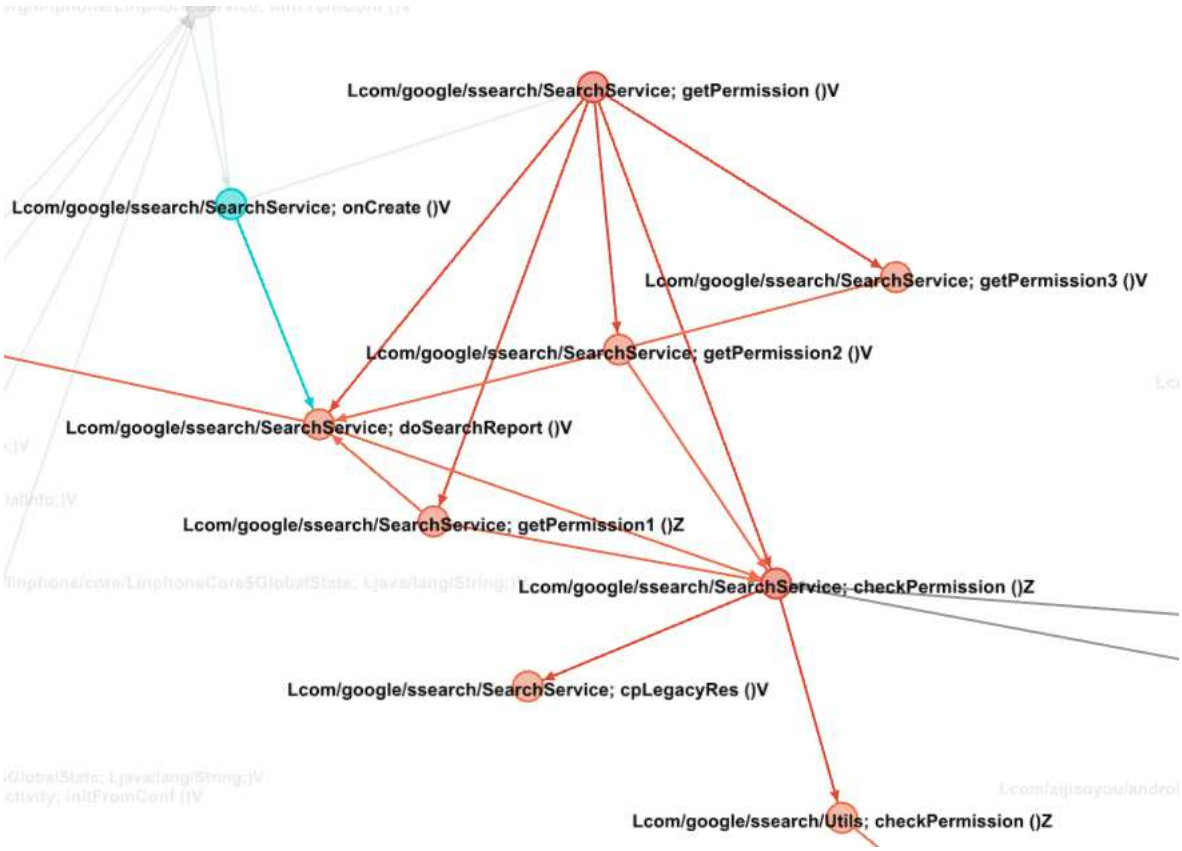


FIGURE A.2 – Extrait du graphe d'appel de fonction d'un échantillon de DroidKungFu

```xml
<application android:label="@string/app_name" android:icon="@drawable/icon">
    <activity android:theme="@style/Theme.Transparent"
        android:label="@string/app_name" android:name=".Main">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <activity android:name=".Help" />
    <activity android:theme="@*android:style/Theme.Dialog"
        android:name="com.google.ssearch.Dialog"
        android:configChanges="keyboardHidden|orientation" />
    <service android:name="com.google.ssearch.SearchService" />
    <receiver android:name="com.google.ssearch.Receiver">
        <intent-filter>
            <action android:name="android.intent.action.BATTERY_CHANGED_ACTION" />
            <action android:name="android.intent.action.SIG_STR" />
            <action android:name="android.intent.action.BOOT_COMPLETED" />
        </intent-filter>
    </receiver>
</application>
```

Listing A.5 – Extrait du contenu du fichier `AndroidManifest.xml` d'un échantillon de DroidKungFu1

```java
public void onCreate() {
    super.onCreate();
    v5 = this.getSharedPreferences("sstimestamp", 0);
    v3 = v5.getLong("start", 0.0, v8);
    v0 = System.currentTimeMillis();
    if(v3 != 0.0) {
        if((v0 - v3) >= 14400000.0) {
            this.mPreferences = this.getSharedPreferences("permission", 0);
            if(com.google.ssearch.Utils.isConnected(this) != 0) {
                this.doSearchReport();
            }
            this.getPermission();
            this.provideService();
        } else {
            this.stopSelf();
        }
    } else {
        v2 = v5.edit();
        v2.putLong("start", v0, v1);
        v2.commit();
        this.stopSelf();
    }
    return;
}
```

Listing A.6 – Méthode `onCreate` du composant `SearchService` d'un échantillon de DroidKungFu1

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <long name="start" value="1410188921456" />
</map>
```

Listing A.7 – Contenu du fichier `sstimestamp.xml` d'un échantillon de DroidKungFu1

A.4 DroidKungFu2

DroidKungFu2 [60] est un malware découvert en 2011 et considéré comme ayant été écrit par le même groupe de développeurs que DroidKungFu1. Le rapport sur son analyse indique qu'ils ont d'ailleurs un comportement similaire car ils exploitent tous deux des vulnérabilités dans le système pour élever leur privilèges et installer de nouvelles applciations sur le téléphone. La différence entre les attaques sont les applications qu'ils installent sur le téléphone. En analysant un des échantillons de DroidKungFu2 [5], nous remarquons rapidement que leur mode d'exécution est le même. Voir listing A.8. Dans DroidKungFu2, le code malveillant est éxécuté lors de la création d'un service et sous la condition qu'un certain temps se soit écoulé. Nous pouvons aussi d'ailleurs remarquer que le nom de certaines méthodes appelées sont exactement les mêmes (ex : `getPermission`).

Il suffit de modifier la valeur du paramètre `start` dans `sstimestamp.xml` pour faire ainsi croire à l'application que plus de $1800000.0ms$ se sont écoulées et forcer l'exécution du code malveillant.

```
1  public void onCreate() {
2      super.onCreate();
3      v5 = this.getSharedPreferences("sstimestamp", 0);
4      v3 = v5.getLong("start", 0.0, v8);
5      v0 = System.currentTimeMillis();
6      if(v3 != 0.0) {
7          if((v0 - v3) >= 1800000.0) {
8              this.mPreferences = this.getSharedPreferences("permission", 0);
9              this.updateInfo();
10             this.getPermission();
11             this.provideService();
12         } else {
13             this.stopSelf();
14         }
15     } else {
16         v2 = v5.edit();
17         v2.putLong("start", v0, v1);
18         v2.commit();
19         this.stopSelf();
20     }
21     return;
22 }
```

Listing A.8 – Méthode `onCreate` d'un composant `service` d'un échantillon de DroidKungFu2

5. MD5 : f438ed38b59f772e03eb2cab97fc7685

A.5 jSMSHider

Les échantillons de jSMSHider exécutent tous leur code malveillant dès le lancement de l'application. Il n'y a ainsi nul besoin d'introduire manuellement des évènements dans le système.

Publications

V Viet Triem Tong, R Andriatsimandefitra, S Geller, S Boche, F Tronel, C Hauser. Mise en œuvre de politiques de protection des flux d'information dans l'environnement Android. *Computer & Electronics Security Applications Rendezvous*, 2011. Rennes, France.

Radoniaina Andriatsimandefitra, Stéphane Geller, Valérie Viet Triem Tong. Designing information flow policies for android's operating system. *IEEE International Conference on Communications*, 2012 (ICC 2012). Ottawa, Canada

R Andriatsimandefitra, V. Viet Triem Tong, L Mé. Diagnosing intrusions in Android operating system using system flow graph. *Workshop Interdisciplinaire sur la Sécurité Globale*, 2013 (WISG 2013). Troyes, France.

M Jaume, R Andriatsimandefitra, V. Viet Triem Tong, L Mé. Secure States versus Secure Executions. *9th International Conference on Information Systems Security*, 2013 (ICISS 2013). Kolkata, India.

R Andriatsimandefitra, V Viet Triem Tong, T Saliou. Information Flow Policies vs Malware. *International Conference on Information Assurance and Security*, 2013 (IAS 2013). Yassmine Hammamet, Tunisia.

R Andriatsimandefitra and V Viet Triem Tong. Capturing Android Malware Behaviour using System Flow Graph. *The 8th International Conference on Network and System Security*, 2014 (NSS 2014). Xi'an, China

R Andriatsimandefitra, V Viet Triem Tong, T Saliou. Information Flow Policies vs Malware – Final Battle - . *Journal of Information Assurance and Security*, Dynamic Publishers Inc., USA, 2014, 9 (2), pp.72-82.

R Andriatsimandefitra, V Viet Triem Tong. Poster Abstract : Highlighting Easily How Malicious Applications Corrupt Android Devices. *Research in Attacks, Intrusions and Defenses*, 2014 (RAID 2014). Gothenburgh, Sweden.

Bibliographie

[1] Android-root 2009-08-16 source. http://www.zenthought.org/content/file/android-root-2009-08-16-source.

[2] Androtótal. http://andrototal.org.

[3] Bionic (software). http://en.wikipedia.org/wiki/Bionic_%28software%29.

[4] Cve-2009-2692. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-2692.

[5] Cve-2013-6282. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-6282.

[6] dedexer. http://dedexer.sourceforge.net/.

[7] droidbox. android application sandbox. https://code.google.com/p/android-apktool/.

[8] F1 score. https://en.wikipedia.org/wiki/F-score.

[9] Ida pro. https://www.hex-rays.com/products/ida/.

[10] Java decompiler. http://jd.benow.ca/.

[11] Netlink protocol library suite (libnl). http://www.carisma.slowglass.com/~tgr/libnl.

[12] Programmation java/réflexion. http://fr.wikibooks.org/wiki/Programmation_Java/R%C3%A9flexion.

[13] smali, an assembler/disassembler for android's dex format. https://code.google.com/p/smali/.

[14] Virustotal. https://www.virustotal.com/.

[15] dm-crypt. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/device-mapper/dm-crypt.txt, 2014.

[16] Radoniaina Andriatsimandefitra, Stéphane Geller, and Valérie Viet Triem Tong. Designing information flow policies for android's operating system. In *IEEE International Conference on Communications*, 2012.

[17] Radoniaina Andriatsimandefitra and Valérie Viet Triem Tong. Capturing Android Malware Behaviour using System Flow Graph. In *NSS 2014 - The 8th International Conference on Network and System Security*, Xi'an, China, October 2014.

[18] Radoniaina Andriatsimandefitra, Valérie Viet Triem Tong, and Ludovic Mé. Diagnosing intrusions in android operating system using system flow graph. In *Workshop Interdisciplinaire sur la Sécurité Globale*, 2013.

[19] Radoniaina Andriatsimandefitra, Valérie Viet Triem Tong, and Thomas Saliou. Information flow policies vs malware–final battle–. *Journal of Information Assurance and Security*, 9(2) :72–82, 2014.

[20] Radoniaina Andriatsimandefitra, Valérie Viet Viet Triem Tong, and Thomas Saliou. Information flow policies vs malware. In *Information assurance and security - 2013*, 2013.

[21] Andrew W. Appel. Deobfuscation is in np, aug 2002.

[22] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin : Effective and explainable detection of android malware in your pocket. In *NDSS 2014*, 2014.

[23] Arini Balakrishnan and Chloe Schulze. Code obfuscation literature survey. Technical report, Computer Sciences Department - University of Wisconsin, Madison, 2005. CS701 Construction of Compilers, Instructor : Charles Fischer.

[24] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. Mockdroid : trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, 2011.

[25] Thomas Blasing, Leonid Batyuk, A-D Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. An android application sandbox system for suspicious software detection. In *Malicious and unwanted software (MALWARE), 2010 5th international conference on*, pages 55–62. IEEE, 2010.

[26] Ohad Bobrov. Chinese government targets hong kong protesters with android mrat spyware. `https://www.lacoon.com/chinese-government-targets-hong-kong-protesters-android-mrat-spyware/`, 2014.

[27] Dan Boneh and Matt Franklin. Identity-based encryption from the weil pairing. In *Advances in Cryptology—CRYPTO 2001*, pages 213–229. Springer, 2001.

[28] Joany Boutet and Tom Leclerc. Grand theft android : Phishing with permission. `http://sagsblog.telinduslab.lu/index.php/post/201309%5CS-Team-at-Hacklu-2013%3A-GRAND-THEFT-ANDROID%3A-PHISHING-WITH-PERMISSION`, 2013.

[29] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. Xmandroid : A new android evolution to mitigate privilege escalation attacks. Technical report, Technische Universitat Darmstadt, Center for Advanced Security Research Darmstadt, 2011.

[30] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. mytunes : Semantically linked and user-centric fine-grained privacy control on android. Technical report, CASED, 2012.

[31] C-Skills. Android trickery. `http://c-skills.blogspot.fr/2010/07/android-trickery.html`, 2010.

[32] C-skills. Yummy yummy, gingerbreak ! `http://c-skills.blogspot.fr/2011/04/yummy-yummy-gingerbreak.html`, 2011.

[33] Victor Chebyshev and Roman Unuchek. Mobile malware evolution : 2013, 02 2014.

[34] Erika Chin, Adrienne Porter Fell, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *MobiSys'11*, 2011.

[35] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

[36] Fred Chung. Custom class loading in dalvik, July 2011.

[37] CIDRE. Blare ids. `www.blare-ids.org`.

[38] CIDRE. Git repositories of blare. `https://git.blare-ids.org`.

[39] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, University of Auckland, 1997.

[40] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, third edition edition, 2005.

[41] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines : And Other Kernel-based Learning Methods*. Cambridge University Press, New York, NY, USA, 2000.

[42] Anthony Desnos and Geoffroy Gueguen. Android : From reversing to decompilation. Technical report, Black Hat Abu Dhabi 2011, 2011.

[43] dex2jar team. dex2jar tools to work with android .dex and java .class files. `https://code.google.com/p/dex2jar/`.

[44] Thomas Eder, Michael Rodler, Dieter Vymazal, and Markus Zeilinger. Ananas-a framework for analyzing android applications. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 711–719. IEEE, 2013.

[45] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazieres, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *ACM SIGOPS Operating Systems Review*, 2005.

[46] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid : An information-flow tracking system for realtime privacy monitoring on smartphones. In *In Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[47] William Enck, Machigar Ongtang, and Patrick Mcdaniel. Mitigating android software misuse before it happens. Technical report, The Pennsylvania State University, 2008.

[48] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear : A library for large linear classification. *The Journal of Machine Learning Research*, 9 :1871–1874, 2008.

[49] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission re-delegation : Attacks and defenses. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11. USENIX Association, 2011.

[50] Eric Filiol. Metamorphism, formal grammars and undecidable code mutation. *J. Comp. Sci*, pages 70–75, 2007.

[51] Jeff Forristal. Android : One root to own them all, 2013.

[52] Linux Foundation. Generic netlink howto. `http://www.linuxfoundation.org/collaborate/workgroups/networking/generic_netlink_howto`, 2009.

[53] Linux Foundation. iproute2. `http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2`, 2009.

[54] Debin Gao, Michael K. Reiter, and Dawn Song. Gray-box extraction of execution graphs for anomaly detection. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 318–329, 2004.

[55] Stéphane Geller. Information flow and execution policy for a model of detection without false negatives. *Network and Information Systems Security (SAR-SSI), 2011 Conference on*, 2011.

[56] Stéphane Geller, Christophe Hauser, Frédéric Tronel, and Valérie Viet Triem Tong. Information flow control for intrusion detection derived from mac policy. In *IEEE International Conference on Communications*, 2011.

[57] Stéphane Geller, Valérie Viet Triem Tong, and Ludovic Mé. Bspl : A language to specify and compose fine-grained information flow policies. In *SECURWARE 2013, The Seventh International Conference on Emerging Security Information, Systems and Technologies*, 2013.

[58] Kent Griffin, Scott Schneider, Xib Hu, and Tzi cker Chiueh. Automatic generation of string signatures for malware detection. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID 2009)*, 2009.

[59] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. "these aren't the droids you're looking for" : Retrofitting android to protect data from imperious applications. In *CCS'11*, 2011.

[60] Xuxian Jiang. Security alert : New droidkungfu variants found in alternative chinese android markets. `http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu2/`.

[61] Xuxian Jiang. Security alert : New sophisticated android malware droidkungfu found in alternative chinese app markets. http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html.

[62] Xuxian Jiang. Questionable android apps – sndapps – found and removed from official android market. http://www.csc.ncsu.edu/faculty/jiang/SndApps/, 2011.

[63] Kevin Kaichuan He. Kernel korner - why and how to use netlink socket. http://www.linuxjournal.com/article/7356, 2005.

[64] Ariane Keller. Kernel space - user space interfaces. http://people.ee.ethz.ch/~arkeller/linux/kernel_user_space_howto.html.

[65] Michael Kerris. The linux programming interfaces - posix message queues. http://man7.org/tlpi/download/TLPI-52-POSIX_Message_Queues.pdf.

[66] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 223–236, Bolton Landing, NY, October 2003. ACM Press.

[67] Christopher Kruegel, Engin Kirda, Paolo Milani Comparetti, Ulrich Bayer, and Clemens Hlauschek. Scalable, behavior-based malware clustering. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS 2009)*, 1 2009.

[68] Mohit Kumar. Dynamic analysis tools for android fail to detect malware with heuristic evasion techniques. http://thehackernews.com/2014/05/dynamic-analysis-tools-for-android-fail.html, 2014.

[69] International Secure Systems Lab. Anubis - malware analysis for unknown binaries. http://anubis.iseclab.org/.

[70] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS '03 : Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, New York, NY, USA, 2003. ACM.

[71] Radmon Llamas, Melissa Chau, and Michael Shirer. Worldwide smartphone market grows 28.6% year over year in the first quarter of 2014, according to idc. http://www.idc.com/getdoc.jsp?containerId=prUS24823414, 2014.

[72] Lisa Mahapatra. Android vs. ios : What's the most popular mobile operating system in your country ? http://www.ibtimes.com/android-vs-ios-whats-most-popular-mobile-operating-system-your-country-1464892, 2013.

[73] Andrew C. Myers and Barabara Liskov. Complete, safe information flow with decentralized labels. In *IEEE Symposium on Security and Privacy*, 1998.

[74] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. *SIGOPS Oper. Syst. Rev.*, 31(5) :129–142, 1997.

[75] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4) :410–442, 2000.

[76] Pablo Neira-Ayuso, Rafael M Gasca, and Laurent Lefevre. Communicating between the kernel and user-space in linux using netlink sockets. *Software : Practice and Experience*, 40, 2010.

[77] Sebastian Neuner, Victor van der Veen, Martina Lindorfer, Markus Huber, Georg Merzdovnik, Martin Mulazzani, and Edgar Weippl. Enter sandbox : Android sandbox comparison. *Proceedings of the IEEE*.

[78] Jon Oberheide and Charlie Miller. Dissecting the android bouncer, 2012. Summercon.

[79] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in android. In *Annual Computer Security Applications Conference*, volume 0, pages 340–349, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[80] Xinming Ou, Wayne F Boyer, and Miles A McQueen. A scalable approach to attack graph generation. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 336–345. ACM, 2006.

[81] PalmSource. Openbinder. `http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html`.

[82] Mila Parkour. Contagio mobile. `contagiominidump.blogspot.com`.

[83] Android Open Source Project. Device administration. `https://developer.android.com/guide/topics/admin/device-admin.html`.

[84] Android Open Source Project. Intent. `http://developer.android.com/reference/android/content/Intent.html#setFlags%28int%29`.

[85] Android Open Source Project. Introducing art. `http://source.android.com/devices/tech/dalvik/art.html`.

[86] Android Open Source Project. Jni tips. `http://developer.android.com/training/articles/perf-jni.html`.

[87] Android Open Source Project. Pro guard. `http://developer.android.com/tools/help/proguard.html`.

[88] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec, April*, 2013.

[89] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 108–125, Berlin, Heidelberg, 2008. Springer-Verlag.

[90] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *J. Comput. Secur.*, December 2011.

[91] Marc Rogers. The bearer of badnews. https://blog.lookout.com/blog/2013/04/19/the-bearer-of-badnews-malware-google-play/, 2013.

[92] Sebastian. Droid2. http://c-skills.blogspot.fr/2010/08/droid2.html, 2010.

[93] Sebastian. Zimperlich sources. http://c-skills.blogspot.fr/2011/02/zimperlich-sources.html, 2011.

[94] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox : having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1808–1815. ACM, 2013.

[95] Tim Starzzere. Security alert : Legacy makes another appearance, meet legacy native (lena). https://blog.lookout.com/blog/2011/10/20/, 2011.

[96] Tim Strazzere. June 15, 2011 security alert : Malware found targeting custom roms (jsmshider). https://blog.lookout.com/blog/2011/06/15/security-alert-malware-found-targeting\discretionary{-}{}{}custom-roms-jsmshider/.

[97] Androguard team. Reverse engineering, malware and goodware analysis of android applications ... and more (ninja !). https://code.google.com/p/androguard/.

[98] Roman Unuchek. The most sophisticated android trojan. https://www.securelist.com/en/blog/8106/The_most_sophisticated_Android_Trojan, June 2013.

[99] Marko Vitas. Art vs dalvik - introducing the new android runtime in kitkat. http://www.infinum.co/the-capsized-eight/articles/art-vs-dalvik-introducing-the-new-android-runtime-in-kit-kat.

[100] Lukas Weichselbaum, Matthias Neugschwandtner, Martina Lindorfer, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis : Android malware under the magnifying glass. Technical report, Vienna University of Technology, 2014.

[101] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 5(2) :32–39, mar 2007.

[102] Chris Wright, Crispin Cowan, and James Morris. Linux security modules : General security support for the linux kernel, 2002.

[103] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security module framework. In *OLS2002 Proceedings*, 2002.

[104] Tim Wyatt. Security alert : New variants of legacy native (lena) identified. http://blog.lookout.com/blog/2012/04/03/security-alert-new-variants-of-legacy-native-lena-identified/, 2013.

[105] Rubin Xu, Hassen Saidi, and Ross Anderson. Aurasium : Practical policy enforcement for android applications. In *USENIX Security '12*, 2012.

[106] Lok-Kwong Yan and Heng Yin. Droidscope seamlessly reconstructing os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Security Symposium*, August 2012.

[107] Jay Yarow. This chart shows google's incredible domination of the world's computing platforms. `http://www.businessinsider.com/androids-share-of-the-computing-market-2014-3`, 2014.

[108] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama : capturing system-wide information flow for malware detection and analysis. In *CCS '07 : Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127, New York, NY, USA, 2007. ACM.

[109] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *OSDI '06 : Proceedings of the 7th symposium on Operating systems design and implementation*, pages 263–278, Berkeley, CA, USA, 2006. USENIX Association.

[110] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *NSDI'08 : Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 293–308. USENIX Association, 2008.

[111] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid : an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 93–104. ACM, 2012.

[112] Yajin Zhou and Xuxian Jiang. An analysis of the anserverbot trojan. Technical report, NQ Mobile Security Research Center, 2011.

[113] Yajin Zhou and Xuxian Jiang. Dissecting android malware : Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.